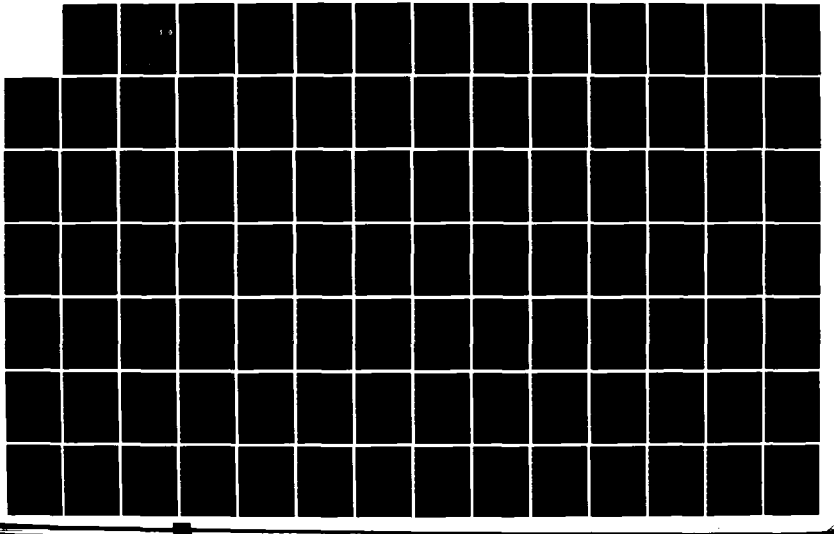
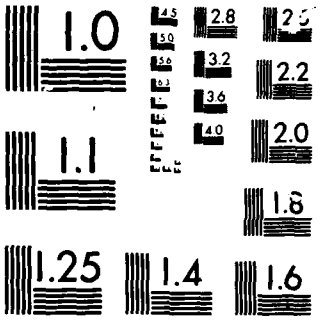
AD-A166 937 A NEW PROGRAM STRUCTURING MECHANISM BASED ON LAYERED GRAPHS(U) STANFORD UNIV CA DEPT OF COMPUTER SCIENCE H L OSSHER DEC 84 STAN-CS-85-1078 MDA903-80-C-0432 1/1

UNCLASSIFIED F/G 9/2 NL





MICROCOPY CHART

December 1984

Report No. STAN-CS-85-1078
Also numbered CIS-G553-1

(2)

A New Program Structuring Mechanism Based on Layered Graphs

by

Harold L. Ossher

Department of Computer Science

Stanford University
Stanford, CA 94305

The research reported in this dissertation was supported in part by the Defense Advanced Research Projects Agency under Contract No. MDA 903-80-C-0432.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-85-1078 CIS-G553-1	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A New Program Structuring Mechanism Based on Layered Graphs		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Harold L. Ossher		8. CONTRACT OR GRANT NUMBER(s) MDA 903-84-K-0062
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford University Stanford, CA 94305 Computer Science Department		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Supply Service - Washington Room 1D-245 The Pentagon Washington, D.C. 20310		12. REPORT DATE December, 1984 13. NUMBER OF PAGES 247
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ONRRR, Stanford University Room 165 Durand Bldg. Stanford, CA 94305		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

This contract is awarded under Basic Agreement MDA 903-84-K-0062, issued by Defense Supply Serivce - Washington. This research is sponsored by Defense Advanced Research Projects Agency (DARPA).

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software engineering, programming methodology, programming languages, structure specification, layered structures, multiple views.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The structure of a computer program and the extent to which that structure is visible dramatically affect the clarity of the program. Programming languages therefore contain *structuring mechanisms* to provide a framework within which to structure programs.

This thesis describes a new program structuring mechanism, called the *grid*.

DD FORM 1 JAN 73 1473

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. The grid mechanism was designed to specify, represent, document and enforce the structure of large programs having a *layered* organisation. Layered programs arise whenever levels of abstraction, layers of protection or multiple views of objects are used, yet they are not handled adequately by other structuring mechanisms.

The structure of a layered program is conveniently modeled by a *layered graph* consisting of interacting layers, each of which is a directed graph of program parts. The grid mechanism is based on this model. A grid specification identifies the layers explicitly, and specifies program structure in terms of them. Similarities between layers are exploited to simplify a specification. Differences between layers, as well as any structural irregularities or relaxation of access restrictions, are highlighted.

The grid mechanism emphasises human readability. It uses some novel techniques to specify layered graph structures in a clear and concise manner, including specification of irregular structures as regular structures with explicit deviations, omission of unnecessary detail, and localisation of information. The grid can specify multiple relationships between program parts. It also serves as a structured repository for information about a program, such as documentation or the information required by source management or version control systems.

A prototype implementation of the grid mechanism was developed, to check the actual structure of Modula-2 programs against grid specifications. Experience with a number of examples, including a large system in widespread use, indicates that the implementation can perform the necessary checks with acceptable efficiency, that grid specifications do not grow excessively large as program size increases, and that the grid mechanism effectively specifies the structure of large, layered programs in a clear and concise manner.

A NEW
PROGRAM STRUCTURING MECHANISM
BASED ON LAYERED GRAPHS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Harold Leon Ossher

December 1984

To the memory of my parents

Yale and Freda Ossher

and to

Ruth and Joel

Acknowledgements

It has been a great pleasure and privilege for me to work with my advisor, Brian Reid. He has been an unfailing source of expertise, whether on technical matters, writing style, document preparation or steak restaurants. I cannot even begin to thank him for all the time he has given me, for his encouragement and assistance, for his good humour,[1] and for his patience.

The members of my reading committee, John Hennessy and Jim Horning, made a number of valuable contributions to this dissertation. They drew my attention to a variety of related research, discussed my ideas with me, and helped me to present them more clearly, accurately and convincingly. I enjoyed my interaction with them, and benefited from it in many ways.

A number of other people discussed my work with me: Danny and Lucy Berlin, Miriam Blatt, Danny Bobrow, Lori Clarke, David Elliott, Stu Feldman, Ivar Jacobson, Mark Linton, Ernst Mayr, Jack Wileden, and Alexander Wolf. Their comments have been most helpful, and more than once led to new insights.

My thanks to those people who helped with the preparation of this dissertation: Danny and Lucy Berlin, Linda Kovach and Ruth Ossher. They saved me valuable time when it mattered most. Special thanks to Linda for providing friendly and outstanding secretarial support all the time I have known her.

[1] British spelling is used throughout this thesis.

During part of my graduate studies, I was supported by a National Scholarship awarded by Rhodes University, Grahamstown, South Africa. I gratefully acknowledge their generosity.

To my office-mates, Lia Adams, Paul Asente, Carolyn Bell and Glenn Trewitt, my thanks for providing a congenial working environment, and for putting up with my increasing ill-humour as deadlines approached. Special thanks to Glenn for writing the program I use most (Vemacs), for keeping the hardware running, and for sneaking an extra half Megabyte of memory into my Sun station.

My wife, Ruth, has been a constant source of love, support, assistance and good food. I am greatly looking forward to being able to spend some time with her again.

Finally, thanks to my baby son, Joel, for his unfailing ability to entertain me, amuse me, delight me, and keep me awake.

The research reported in this dissertation was supported in part by the Defense Advanced Research Projects Agency under Contract No. MDA 903-80-C-0432.

Table of Contents

Chapter 1
Introduction

The structure of a computer program and the extent to which that structure is visible dramatically affect the clarity of the program. Clarity, in turn, affects other important attributes such as reliability and modifiability. For this reason, structure and its specification are of central importance to programming.

Computer programs consist of collections of *parts*, such as modules, procedures, statements and expressions. The precise nature of these parts depends on the programming language being used. The *structure* of a program is the *organisation* and *interactions* of its parts.

Programming languages contain *structuring mechanisms* to provide a framework within which to organise program parts and to specify their interactions. One of the primary purposes of a structuring mechanism is to document structure in a clear and concise manner. Equally important, the structuring mechanism must enforce the documented structure, ensuring that the program does indeed conform to the structure specified. A structuring mechanism should also be sufficiently rich to cope with the wide diversity of program structures that arise in large systems.

Structuring mechanisms are traditionally able to handle programs that are hierarchically organised, or programs that consist of amorphous collections of modules. Examples are *static nesting*, which is widely used for specifying hierarchical structures, and *visibility control lists* (import/export lists), which

control interactions between the modules in an amorphous collection. There is a third form of program organisation that is important and useful, but for which no structuring mechanism exists: *layered* organisation, in which a program consists of a number of separate but interacting *layers*, each of which has its own internal structure. This thesis introduces and defines a new structuring mechanism, called the *grid*, that is especially suited to specifying the structure of layered programs.

The grid mechanism was designed as part of *Fable*, a language for specifying integrated circuit fabrication processes [Ossher-Reid 83]. Such processes, as well as the fabrication equipment that performs them, are described at multiple levels of abstraction. At the same time, each step of a process and each piece of equipment is complex and structured, and is treated as an object. A Fable program is thus an *object-oriented* program in which objects are described at multiple levels of abstraction. A natural and convenient model for the structure of such a program is a *layered graph*, in which each layer corresponds to an abstraction level. Layered graph structures occur whenever levels of abstraction are used to guide system design and implementation; areas in which this is common include digital systems and communication protocols.

Layered structures also arise when objects are used in different ways by different clients. In such cases, it is convenient to regard each client as having its own *view* of each object it uses. The layers then correspond to views, with each layer containing descriptions of many objects, all from a single point of view. Layers corresponding to levels of abstraction are generally ordered, whereas layers corresponding to alternative views might not be ordered; the layered graph model is applicable in both cases. The grid mechanism is based on this model, and is able to specify arbitrary layered graph structures.

When specifying structure, there is often tension between accuracy and readability. Accurate specifications of complex structures tend themselves to be

complex, and therefore difficult to read. One of the most important functions of a structuring mechanism is to document structure, and such documentation is particularly valuable if a new reader who is completely unfamiliar with a program can gain an appreciation of its overall structure without having to examine excessive detail. The technique of *hierarchical decomposition* facilitates this to some extent, by allowing the reader to select how much detail to examine. It is often inappropriate, however, as few large systems conform to a rigid hierarchical structure, though many are nearly hierarchical.

The grid mechanism emphasises documentation of structure, and addresses the issue of specifying complex structures in a readable fashion. It uses four techniques for reducing complexity: *factorisation*, *clustering*, *deviation* and *approximation*.

The techniques of factorisation and clustering both involve collecting closely related nodes together and then treating all nodes in a collection uniformly, replacing the interactions between individual nodes by interactions between the collections. This process has much in common with hierarchical decomposition; the technique of clustering is, in fact, a form of hierarchical decomposition.

The technique of deviation is the key to the success and generality of the other techniques. Though treating collections of nodes uniformly often results in great simplification, it also often results in inaccuracy: the nodes in a collection are in fact different, and might behave differently in some respects. Deviation involves explicit specification of the manner in which individual nodes deviate from the assumed uniform behaviour of collections. It allows a clear, simple overview to be presented, together with specifications of how the actual structure differs from the overview.[2] It allows a nearly hierarchical program to

---

[2]Traditional hierarchical decomposition achieves the same goal to some extent, but low-level details sometimes have to intrude at higher levels to preserve accuracy. Deviation allows such details to be specified separately, without complicating the overview.

be specified as an hierarchy, with deviations specified separately. It allows an unfamiliar structure to be specified as a similar but familiar one, together with a description of how they differ. The deviations often contain the most valuable information about program structure, because they specify those aspects that are exceptional and non-intuitive.

The technique of approximation exists in recognition of the fact that not all deviations are material in all contexts. It allows specifications of deviations to be ignored when not needed, giving rise to approximate specifications of structure that are simpler yet sufficiently accurate. A new reader can begin with a simple but gross approximation, and then proceed to consider further details only as needed.

The grid mechanism has a number of additional important features. A single grid can specify multiple kinds of relationships between the parts of a program. Multiple grids can be used to specify the structure of the same program from multiple points of view. Though designed as part of Fable, the grid mechanism is completely language-independent; in fact, it is not restricted even to programs, and can be used to specify the structure of any collection of entities. A grid is textually separate from the program whose structure it specifies, so it can be created before the program is written, and then used to check that the developing program conforms to the specified structure at all times. It also provides a structured repository for information about the program, such as documentation or information for use by source maintenance or version control systems. These features enhance the usefulness of the grid, though its chief advantage remains its ability to specify layered graph structures in a concise and readable fashion.

This thesis motivates the need for the grid mechanism, defines it mathematically, and describes a prototype implementation for representing and enforcing the structure of Modula-2 programs. The remainder of this chapter

gives background information about structuring mechanisms and Fable, discusses the requirements of a structuring mechanism for Fable, describes some existing structuring mechanisms in the light of these requirements, and motivates the need for a new mechanism. Chapter 2 describes layered programs and the layered graphs used to model their structure, and gives details of the four techniques for specifying layered graph structures in a readable fashion. It also introduces an important example that is used consistently throughout subsequent chapters. Chapter 3 gives an overview of the grid mechanism itself, followed by precise definitions of its abstract syntax and semantics. Chapter 4 describes the prototype implementation of the grid mechanism for Modula-2. Chapter 5 describes three example grid specifications, including one of a large, widely-used program; extracts of actual computer input, output and intermediate forms associated with one of these examples are included as Appendix I. Chapter 6 discusses directions for further research and a number of ideas that have not yet been explored in detail. Finally, Chapter 7 summarises the features of the grid mechanism and shows that it does indeed meet its requirements.

1.1. Background: Structure and Structuring Mechanisms

This section introduces some important concepts and terminology associated with structure and structuring mechanisms.

Structuring mechanisms specify program structure by specifying the organisation and interactions of program parts. They can serve a variety of purposes, including documentation, representation and enforcement of structure, controlling visibility, and aiding program design and construction. The relative importance of these various purposes depends on the environment and manner in which a structuring mechanism is to be used, and greatly affects the design of the mechanism. Those of importance to this research are discussed in detail in Section 1.3.

The parts of a program can vary greatly in size and complexity, and some can contain others; for example, modules can contain procedures, which in turn can contain statements. Any structuring mechanism must consider certain parts to be *atomic*, and specify the structure of the program in terms of those atomic parts. The term *unit* is used throughout this thesis to denote such an atomic part. The nature of the units determines the *granularity* of the structuring mechanism; the smaller the units, the finer-grained the mechanism. A mechanism that treats entire modules as atomic is sometimes referred to as a *module interconnection language* [DeRemer-Kron 76].

1.1.1. Organisation

Programs can be classified, from a structural point of view, according to the manner in which their units are organised. Structuring mechanisms can be classified according to the categories of program organisation they can handle.

Although a great many ways of organising the units of a program can be imagined, three emerge as prominent based on usage in practice:

Amorphous.	Units form an unorganised collection.
Hierarchical.	Units are classified into an hierarchy.
Layered.	Units are partitioned into layers, each of which has its own internal structure.

The amorphous approach tends to be followed when a program is viewed as a collection of separate and largely independent units that can be pieced together and used as needed. It is common in Fortran [Golden 65], Lisp [Teitelman 78] and CLU [Liskov, *et al.* 81] programs, which consist of amorphous collections of subprograms, functions and clusters, respectively, all of which are available for general use.[3] The problem with this organisation is that it does not explicitly identify coherent groups of closely related units.

---

[3] The CLU *library* does allow hierarchical classification of units, however.

As pointed out by Parnas, the term "hierarchy" is used in many different ways, to the extent that "hierarchically structured" actually means very little [Parnas 74]. Throughout this thesis, I shall use term *hierarchy* as synonymous with *tree structure*: programs that are hierarchical according to this interpretation most commonly arise as a result of textual nesting in Algol-like languages. Units in an hierarchical program are grouped according to common features, such as common function, common purpose or common attributes; the common features of a collection of children are associated with and characterise their parent. This kind of grouping is natural and intuitive, and corresponds to a large extent to the way in which people structure their thoughts and concepts. Hierarchical organisation is extremely common in computer programs, as elsewhere. It is supported by languages that allow static nesting, such as Algol60 [Naur 63], Pascal [Jensen-Wirth 78] and Ada [ANSI 83], by object-oriented languages that include class hierarchies, such as Simula 67 [Dahl-Myhrhaug-Nygaard 68] and Smalltalk [Ingalls 78], and by a variety of structure specification languages, such as MIL 75 [DeRemer-Kron 76], Gandalf SVCL [Kaiser-Habermann 82], C/Mesa [Mitchell-Maybury-Sweet 79] and SML [Schmidt 82].

A layered program consists of a number of *layers*, each of which consists of a collection of units organised in some other way. Layers commonly represent *levels of abstraction* or *viewpoints*: ideally, the units in a layer describe entities at a consistent level of abstraction or from a single, consistent point of view. When layered structures arise from levels of abstraction, the layers are generally ordered on the basis of degree of abstraction; when they arise from the presence of multiple viewpoints, there might be no such ordering. Horning and Randell give a lucid description and analysis of layered structures, among others, in their paper on process structuring [Horning-Randell 73]. Layered organisations are common in such fields as digital system design and communications protocols. Some examples from another area, operating systems, are the "THE" Operating System [Dijkstra 68] and MULTICS [Project MAC 74]; in the latter, the layers

are referred to as "rings of protection", and have a complex dynamic protection scheme built on top of them. The structure of the "THE" system is often considered to be hierarchical as well as layered; Parnas discusses this issue in detail [Parnas 74]. Layered programs are not always hierarchical, however; Section 1.6 gives an example of one that is not.

1.1.2. Interactions

For the purposes of this thesis, any relationship of interest between a pair of units is considered to be an *interaction* between those units.[4] Many different relationships between units, and hence many different kinds of interaction, are possible. For example, one unit might refer to a type defined in another, invoke a procedure defined in another, be a subclass or superclass of another, etc. The most common kind of interaction is "references", or "uses": unit *a* references or uses unit *b* if and only if the text of *a* somewhere mentions *b*. This kind of interaction subsumes most of the others, and is often the only one used for specifying program structure.

Permitted interactions are often characterised by means of *visibility*: if one part of a program is visible to another, according to the rules of the programming language, then the two parts can interact. The concept of visibility, however, implies that different kinds of interaction between program parts are not differentiated; it is suitable only for specifying structure in terms of references, where different kinds of references are not distinguished. A more general model than visibility is needed to characterise multiple kinds of interactions. The one used throughout this thesis is the *relational model*: each kind of interaction is characterised as a separate *relation* on units. Many such relations may be useful in specifying the structure of a single program.

---

[4] This is a broad definition of the term. It is appropriate here because any kind of relationship between units can provide valuable structural information.

There are four fundamental approaches to specifying what interactions are permitted between the units of a program:

No Restriction.	Arbitrary interactions between units are permitted.
Implicit.	Permitted interactions are derived entirely from the organisation of the units.
Explicit.	Permitted interactions are specified explicitly; any interaction that is not specified is not permitted.
No Interaction.	The units are independent.

Structuring mechanisms can be classified according to which of these approaches they use.

The most common example of implicit derivation is the Algol60 scope rules [Naur 63]. They derive permitted interactions from hierarchical structure as follows: a unit can access only its children, its siblings, and its ancestors and their siblings. This approach is appealing because specification of organisation and interactions are so closely related, and in a natural and intuitive manner. Unfortunately, this close relationship often leads to difficulties: a logical program organisation might permit either too many or too few interactions, and tampering with the organisation so as to permit a specific set of interactions is invariably inelegant and often impossible.

Explicit specification of interactions is general and powerful, allowing arbitrary interactions to be specified accurately. It is epitomised by the mechanism of import lists in such languages as Ada [ANSI 83], Mesa [Mitchell-Maybury-Sweet 79] and Modula-2 [Wirth 83], or export lists (called *access lists*) in Gypsy [Ambler, *et al.* 77]. Such specifications can make use of organisational information, and can use interactions derived automatically from program organisation as defaults to handle common cases.

1.2. Background: Fable

The research described in this thesis was motivated by the need for a structuring mechanism for the Fable language [Ossher-Reid 83]. A brief discussion of the purpose and nature of Fable follows, to give insight into the requirements of the mechanism.[5]

Fable is being developed as a *recipe language* for describing integrated circuit fabrication processes. The purpose of a Fable program is to specify an entire fabrication process and all the equipment it uses, in sufficient detail and with sufficient precision that it can be used to control actual fabrication. The following fundamental requirements of the recipe language were identified early in the design effort:

- *Completeness.* It must be possible to express any conceivable recipe in the language. This should hold true even with dramatic and unforeseen changes in technology.
- *Readability.* A recipe must be clear and easy to follow, yet complete, rigorous and unambiguous. This is particularly important as the readers of recipes will be fabrication engineers, not computer scientists.
- *Safety.* Fabrication processes make use of deadly gasses, high temperatures and delicate equipment. It must not be possible for an error in a recipe to lead to disaster.
- *Portability.* It is particularly important that a recipe developed at one site can be used at another, similar site. Portability through time is equally important: it must be possible for a recipe developed now to be used years in the future, despite changes in technology.
- *Suitability for processing by other programs.* The system as a whole is intended to contain expert systems and other programs that assist engineers in designing recipes and in compensating for errors. These programs must be able to treat recipes as data in a convenient manner.

The means by which we hope to satisfy these requirements are discussed in

---

[5] Fable is still under development. The following description is sufficiently general, however, that it is likely to remain valid even as details of the language change.

detail in the paper introducing Fable [Ossher-Reid 83]. Only structural aspects of the language are considered here.

Because of the complexity of modern integrated circuit fabrication processes, a Fable program that completely describes a real process is expected to be vast and complex. To make the size and complexity manageable, and for reasons of portability, processes and equipment are specified at multiple levels of abstraction, called *views*:

- The *process view*. This is just the name of the process, and details of any parameters it takes.
- The *material view*. This level consists of operations described in terms of their effect on the raw material being processed, without reference to specific fabrication equipment. An example of such an operation is "put a coating of oxide on a wafer".
- The *abstract equipment view*. This level describes all the fabrication equipment used by the process in abstract terms, identifying the operations that can be performed by each piece of equipment. A furnace, for example, is specified as a piece of equipment that can "heat up", "bake", and so forth.
- The *interface-independent view*. This level describes all details of equipment that are not dependent on whether the equipment is controlled manually by technicians or automatically by a computer system. A furnace, for example, consists of various components, including a temperature controller and several gas controllers.
- The *technician view*. This view describes each piece of equipment as seen by technicians who control it manually, giving details of all knobs, gauges, buttons, lights, etc. that form the technician/equipment interface. The temperature controller of a furnace, for example, might consist of three control knobs and three gauges: one knob and gauge for each end of the furnace tube, and one for the centre.
- The *system view*. This view describes each piece of equipment as seen by a computer system that controls it automatically, giving details of all signals that can be sent and all information that can be requested.

A Fable program therefore has *layered* organisation, with each layer corresponding to a view. The technician and system views are alternatives at the same level of abstraction.

In the domain of fabrication, where one is dealing with physical objects, it is natural to adopt an *object-oriented* style of progamming [Dahl-Myhrhaug-Nygaard 68, Ingalls 78, Bobrow-Stefik 83]. Fable is based upon this style. Each piece of fabrication equipment is an object, as is each of its components and subcomponents. Each piece of material that is processed, such as a wafer, is an object. Each type and value used in a Fable program is an object. The fabrication facility itself is an object that contains all the others. The objects in a Fable program can interact in a variety of ways. From this point of view, therefore, a Fable program is a large collection of interacting objects.

Fable programs are thus both *layered* and *object-oriented*. Such programs also occur in other domains, and present some special problems from the point of view of structure specification. Layered, object-oriented programs are described in detail in Section 2.1.

Constructing, maintaining and understanding a Fable program, like any large program, is difficult. Understanding its structure is a crucial first step, but is itself a complex task. There is a clear need for a structuring mechanism to provide assistance with this task. The next section discusses the specific requirements of the mechanism; the rest of this thesis describes the actual mechanism that was developed to meet these requirements.

1.3. Research Goals

The nature of Fable, and the manner and environment in which it is to be used, dictate the following key requirements for its structuring mechanism:

- *Specification*. It must facilitate the explicit specification of program structure by human beings involved in the development of programs.
- *Documentation*. The specification must be clear and readable, and act as an aid to readers in understanding the program.
- *Enforcement*. The specified structure must be enforced automatically, to ensure both that the specification is accurate and that unintended interactions are not introduced into the program.

- *Representation.* The specified structure must be represented in a form suitable for use by software that is concerned with program structure, such as a browser, editor, compiler or interpreter.
- *Ability to handle layered, object-oriented programs.* The mechanism must be able to handle layered, object-oriented programs well, explicitly identifying the layers and objects and specifying their interactions.
- *Scale.* The mechanism must be able to handle large programs, and structure specifications must not grow unmanageably large and complex as program size increases.

Each of these is discussed in detail below, first in general and then with specific reference to Fable.

The goal of the research described in this thesis was to develop a structuring mechanism that meets the above requirements.

1.3.1. Specification

There are two approaches to making the structure of a program apparent: *specifying* it in a suitable fashion, or deriving a suitable description automatically by *analysing* the program itself. The specification approach is followed by structuring mechanisms; the analysis approach is followed by analysis and display tools, such as Masterscope [Teitelman-Masinter 81], and by software databases, such as OMEGA [Linton 83]. There are advantages to both approaches, and there is place for both in a single programming environment.

The fundamental advantage of the specification approach is that it provides a means to communicate structure between people. The designer of a program will generally have a mental model of the program's structure. That model affects his understanding of the program, his design decisions, and ultimately the details of the program itself. A reader of the program, particularly a reader unfamiliar with it, is likely to derive great benefit from knowing the structural model that the program designer used. A structure specification allows the designer to communicate this model to all readers. This argument is similar to

an argument against automatic type assignment in polymorphic languages attributed to Hoare: since the author of a program knows the type of each variable, it is senseless to throw that information away and then attempt to deduce it automatically. DeRemer and Kron argue similarly:

> An MIL [module interconnection language] should provide a means for the programmer(s) of a large system to express their *intent* regarding the overall program structure in a concise, precise and checkable form. [DeRemer-Kron 76]

Another advantage of the specification approach is that the added redundancy introduces the possibility of added control and safety. A program can be checked against its structure specification to determine the validity of all interactions. A structure specification can aid program development and maintenance: the structure can be designed and specified prior to or in parallel with program development, and the specification used as a means of communication among team members and from designers to implementors to maintainers. A structure specification can even be used to control program development if it contains *sufficient supplementary information about* responsibility, deadlines, and so forth: it can prevent unauthorised read or write access to program units, can help monitor development progress, and can ensure at all times that only authorised interactions actually occur between program units. These safeguards are not available if the analysis approach is used.

The fundamental advantages of the analysis approach are two-fold: it frees the program designer from the burden of writing a structure specification and updating it as the program changes, and it allows the structure to be examined in different ways and from different points of view. This is a more dynamic approach, allowing the reader to control the particular viewpoint he wishes to take.[6] Unfortunately, the option of taking the designer's viewpoint is seldom available, as it cannot, in general, be derived accurately from the program.

---

[6] Many of the structural views that can be derived directly from a program can also be derived from a specification of the structure of that program. A browser for examining structure specifications can therefore provide many of the facilities of analysis tools.

Although, in general, there is room for argument as to which approach is preferable, the primary importance of safety in the Fable environment strongly indicates the specification approach.

1.3.2. Documentation

The structure of a large program can be exceedingly complex, yet an appreciation of it is vital to an understanding of the program. It is, in fact, so important, that one generally tries to understand the overall structure before examining the details. Understanding the structure helps to put the program into perspective, introduces one to its parts, and helps to direct one's attention to the parts one needs to examine in detail. The many advantages of structured programming derive from the fact that good structure is an important aid to human beings in understanding and managing complexity [Dijkstra 72]. They apply only if the structure is readily apparent. Structuring mechanisms can aid significantly in making structure apparent and in communicating it between people. They therefore have an important effect on program readability.

Two alternatives have a substantial effect on the value of a structure specification from the point of view of documentation:

- *Permitted* versus *actual* interactions.
- *Global* versus *local* perspective.

When specifying interactions between units, one can specify either interactions that one considers permissible and that might potentially occur in the program, or interactions that actually occur in the program. When engaged in program design, one is likely to be most interested in permitted interactions: as one is composing a unit, one can determine what interactions it is permitted to engage in. Permitted interactions tend to convey designer's intent. Actual interactions, on the other hand, reflect the current state of the program. They are particularly important to systems, such as compilers and version control systems, rather than to human readers, and they can be derived from the

program in a straightforward manner. Permitted interactions are therefore more important from the documentation point of view.

Structure can be examined from either a *global* or a *local* perspective. When examining the structure of a program from the global perspective, one is interested in gaining an appreciation of the overall structure, in finding out the major parts of the program and how they interact. This perspective is especially applicable to a reader unfamiliar with the program, who is trying to gain an understanding of it. When dealing with a particular part of a program, on the other hand, one tends to be interested in how that part interacts with other parts; this is the local perspective. It is especially applicable to a programmer working on a single part of a program, or to a system attempting to determine what to recompile when a particular part of the program changes. Local perspectives can generally be derived from a comprehensive global specification, or from the program itself, but not vice versa (see Section 1.3.1). Thus, from the point of view of documentation, the global perspective is superior.

It is important to note that readability of a structure specification is a notational rather than a semantic issue. A variety of notations for specifying structure might be semantically equivalent, in that they are able to specify the same structures accurately, yet might differ widely in how intelligible they are to human readers. This situation is similar to the "Turing tarpit" of programming languages: though programming languages are Turing-equivalent and hence can specify arbitrary algorithms, they differ widely in how naturally they express the algorithms that arise in diverse domains. Hoare has written of programming languages that:

> The most valuable feature of a programming language is that it provides the programmer with a conceptual framework which enables him to think more clearly about his problems and about effective methods for their solution; and it gives him a notational technique which enables him to express his thoughts clearly. [Hoare 68]

This remark is equally true of structuring mechanisms.

As an illustration of the importance of notation, imagine a programmer presented with a program he has never seen, and a set of ordered pairs representing interactions between the parts of the program. Such a set of pairs is an excellent means of specifying interactions from a semantic point of view: it captures all the relevant information in a simple and elegant manner. Yet the first thing the programmer is likely to do it is to construct a diagram from the set. The first diagram to emerge is likely to be a directed graph, with each edge corresponding to a pair in the set. This diagram is nothing other than a graphical representation of the set of pairs, and can be thought of as a mere syntactic variation, but it is an important one because it is much more readily apprehended by human beings. If the set of pairs is very small, such a diagram might suffice. Otherwise it quickly becomes too complex to make any sense, whereupon the programmer resorts to other techniques. Common ones are rearranging the nodes so as to reduce the number of intersecting lines, and grouping the nodes into clusters. These processes are greatly enhanced by an understanding of what the nodes represent, such as might be obtained from comments in the program. Even such diagrams might become too complex, in which case they might be split into multiple diagrams, have textual annotations, labels and labelled arrows added, and so on. The end result will look much different from the set of ordered pairs, though the essential information it represents is the same. It is likely to be much easier for another human being to understand; anyone else given the program, the set of pairs and the diagrams is likely to examine the diagrams first, and, if they are accurate and complete, may never look at the set of pairs at all. The difference between the diagrams and the set of pairs is purely notational, yet the difference in their readability is vast.

Documentation of structure is of primary importance in the Fable environment because of the readability, safety and portability requirements of Fable. An accurate understanding of program structure is particularly important when any modification is to be made, whether to change the effect of the program or to change the fabrication equipment on which it can run, because

such understanding reduces the risk of unintended side-effects. In a fabrication environment, such side-effects are no mere nuisance; they can be costly and even dangerous.

1.3.3. Enforcement

The utility of a structure specification increases enormously if the specified structure is enforced automatically. The primary reasons are as follows:

- *Consistency.* Enforcement ensures that the specification of structure and the actual structure of the program are consistent. This is in contrast to most forms of documentation, where enforcement is not possible and maintaining consistency is a major problem. It means that a reader can examine the structure specification with confidence that it truly reflects the structure of the program.
- *Debugging aid.* Unintended interactions can be trapped before they give rise to strange runtime errors.
- *Control.* Unauthorised interactions can be trapped, preventing a programmer from violating interface conventions established by the project leader. A good example of this use of enforcement is the separation of definition (specification) from implementation in many modern languages, such as CLU [Liskov, *et al.* 77], Ada [ANSI 83], Mesa [Mitchell-Maybury-Sweet 79] and Modula-2 [Wirth 83]: a unit can use only the facilities made public in the definition part of another unit, and is never allowed access to details of its implementation.

Even apart from all its other advantages, enforcement is an absolute requirement in the Fable environment because of the importance of safety during fabrication.

1.3.4. Representation

A structure specification can be a valuable representation of program structure, of use to software that manipulates programs. Such software includes:

- *Browsers.* Browsers allow a human reader to view portions of a program, and to move from one portion to another. Such motion between portions is largely governed by program structure. A notable

example is the browser within the PIE system [Goldstein-Bobrow 80a, Goldstein 81].

- *Program development aids.* If the structure of a program is determined and specified early, the programming environment can monitor the program development process to ensure that the developing program conforms to the specified structure. If additional information is added to the structure specification, the environment can also restrict access to units under development and assist in co-ordinating the development effort. This aspect has been explored in MIL 75 [DeRemer-Kron 76] and Gandalf [Habermann, *et al.* 82].
- *Version control systems.* Version control systems make heavy use of program structure, and are based upon representations of program structure. Notable examples are Make [Feldman 79], Gandalf [Kaiser-Habermann 82], PIE [Goldstein-Bobrow 80a] and SML [Schmidt 82].

The use of a structure specification as a representation of structure is, of course, dependent on its being machine-readable.

Fable programs are required to be suitable for processing by other programs. To fulfil this requirement, it is necessary that structure specifications of Fable programs be suitable for such processing also. That way, software operating on Fable programs can obtain any needed structural information directly from the structuring mechanism, without having to derive it from the program.

1.3.5. Ability to Handle Layered, Object-Oriented Programs

The units of a layered, object-oriented program can be categorised according to two criteria: layer and object. All the units in a layer, describing different objects, but from a single point of view, form a cohesive group. At the same time, all units describing a single object, but from different points of view, also form a cohesive group. This dual categorisation of units is an important aspect of the structure of a layered, object-oriented program; it is illustrated and discussed in detail in Chapter 2.

Structuring mechanisms traditionally categorise units in only one way, usually as an hierarchy. Handling dual categorisation poses interesting challenges for structuring mechanisms, including:

- Identifying the layers and specifying how they are related.
- Identifying the objects and specifying how they are related.
- Identifying correspondences between units in different layers. This includes identifying the units in the various layers that correspond to each object, thereby making the dual categorisation explicit.
- Identifying any similarity of structure between different layers. Since the layers represent different views of the same objects, there is reason to expect at least some similarity. Specifying it explicitly is a great aid to understanding structure.

The layered, object-oriented programming style is an integral part of Fable. It is therefore imperative that the structuring mechanism for Fable be able to handle adequately the dual categorisation of units.

1.3.6. Scale

DeRemer and Kron introduced the notion that structuring large programs is a substantially different activity from writing small programs [DeRemer-Kron 76]. They analyse the problems inherent in structuring large programs, and list the requirements of a *module interconnection language*. The primary problem in dealing with large programs is the management of complexity: many techniques that are successful in dealing with small programs break down in the face of the complexity of large programs.

The anticipated size and complexity of Fable programs makes scale a critical issue. The structuring mechanism for Fable must provide as much assistance as possible with developing, maintaining and understanding large programs. It must also be such that the structure specification itself does not grow unmanageably large and complex as program size and complexity increase.

1.4. Prior Work

This section describes important structuring mechanisms that existed prior to the development of the grid, in the light of the requirements identified above. Some of the mechanisms described are programming language constructs, some are explicitly intended as mechanisms for specifying structure, some form the basis of version control systems.

1.4.1. Nesting

Nesting schemes facilitate the specification of hierarchical structures, by allowing children to be textually enclosed within their parents. Permissible interactions are derived from the hierarchical organisation, using the Algol60 scope rules or a variant of them in which the additional rule is added that a unit can never access other units that appear after it in the program text. Nesting has been in widespread use since the advent of Algol60, and is provided in many modern languages, such as Pascal, Ada, Mesa and Modula-2.

The main advantages of nesting are that it can specify the hierarchical structures so common in modern programming, and that it is easy to implement and to explain. Nesting as a means of documenting hierarchical structure is reasonable, but not ideal. In long programs with many levels of nesting it is difficult to discern the structure by examining the code, particularly because of the wide spatial separation of different parts of the same construct needed to make way for subsidiary constructs. The chief disadvantage of nesting schemes, however, is that is often impossible to restrict interactions adequately. Clarke, Wileden and Wolf demonstrate this clearly, argue that modules and current thoughts on programming methodology have rendered nesting obsolete, and advocate a nest-free program style for Ada [Clarke-Wileden-Wolf 80]. It is certainly true that, though nesting is useful in some circumstances, it is not adequate in general.

1.4.2. Visibility Control Lists

Many programming languages require interactions between units to be specified explicitly by means of import and/or export lists, referred to here as *visibility control lists.* Examples of such languages are Ada, Mesa, Modula-2 and Gypsy. Gypsy uses export lists, called *access lists*, whereas the other three use import lists, called *with clauses* in the case of Ada. Visibility control lists make no statement about program organisation, and so treat programs as amorphous collections of units.

The nature of visibility control lists varies from language to language, and they are not always detailed enough to facilitate precise specification of interactions. Wolf, Clarke and Wileden have developed a formalism for describing and evaluating visibility control mechanisms [Wolf-Clarke-Wileden 83]. They define the concepts of *accessibility* and *provision*, and measure the *preciseness* of structuring mechanisms on the basis of how accurately they can specify both accessibility and provision. They use their formalism to evaluate a few mechanisms, including nesting, Ada's *with* clauses and Gypsy's *access lists*. They also propose language constructs for precise interface control that are, in effect, detailed visibility control lists [Clarke-Wileden-Wolf 83]. They are currently developing a support environment consisting of a number of tools for managing libraries, creating and modifying modules, and analysing structure.

The chief advantage of visibility control lists is that, provided they are sufficiently detailed, they can specify and enforce arbitrary structures. They are also an excellent means of specifying structure from the local perspective: the import list of a unit directly specifies all other units it uses, and the export list directly specifies all other units it is used by. In small programs, visibility control lists are also adequate for providing the global perspective. Their chief disadvantage is that they fail to provide an adequate global perspective of the structure of large programs. The lists are scattered throughout the program and

define an interaction graph of potentially immense complexity, and since they deal with amorphous programs there is no logical grouping to assist the user in dealing with the large number of units.[7]

1.4.3. MIL 75

In their classic paper on programming-in-the-large, DeRemer and Kron proposed a module interconnection language, called MIL 75, for specifying the structure of large systems [DeRemer-Kron 76]. The basis of a MIL 75 structure description is a *system tree* defining a system hierarchy. The tree is augmented by resource information, accessibility information and modules. The resource information lists the resources required of and actually provided by each system in the hierarchy. The accessibility information specifies permitted interactions between systems in the hierarchy; default rules cover common cases. The modules are the actual units making up the program and responsible for providing all the resources; they can be written in any programming language(s).

MIL 75 thus combines hierarchical organisation with explicit specification of interactions. This is a powerful combination, for it provides the advantages of hierarchical decomposition without being subject to the limitations of nesting. Unfortunately, MIL 75 does not allow arbitrary interactions to be specified: the specifications are closely tied to the hierarchical structure, and are governed and restricted by it to a large extent.

MIL 75 was intended to serve as a project management tool, a design tool, a means of communicating between programmers, and a means of documenting structure. It takes the global perspective, and does a good job of making the overall structure of a program apparent. It also explicitly supports the important

---

[7] Automated tools for analysing and displaying structure, such as Masterscope [Teitelman-Masinter 81], can assist in making sense of this complexity, by attempting to derive a global perspective. This approach was discussed in Section 1.3.1.

techniques of stepwise refinement [Dijkstra 72], information hiding [Parnas 71], and the construction of virtual machines [Dijkstra 72]. It has no mechanism for identifying layers explicitly, however, and so is not suited to layered programs that are not also hierarchical.

1.4.4. DF Files and SML

The culmination of recent work in the area of version control and automatic system maintenance is the work of Lampson and Schmidt [Schmidt 82, Lampson-Schmidt 83] on controlling the software development of Cedar [Deutsch-Taft 80]. They refer to various previous systems, such as Make [Feldman 79], SMF [Cristofor-Wendt-Wonsiewicz 80], Gandalf [Cooprider 79, Tichy 80, Kaiser-Habermann 82] and C/Mesa [Mitchell-Maybury-Sweet 79]. Only the Lampson-Schmidt work is discussed here, however, as it subsumes the structure specification facilities of all the others.

Lampson and Schmidt developed two languages for describing interconnections between Cedar modules in large systems: simple *description files* (DF files) and the much more sophisticated system modeling language, SML. A DF file simply lists all files making up a system or sub-system. Various kinds of files, such as source, object, documentation and command files, are included, and specific locations and versions are specified. A number of useful tools process the files listed in a DF file, checking them for version consistency, releasing them for general use, and so on.

As mentioned by Schmidt, DF files contain little structure [Schmidt 82]. Their most important structural property is that one DF file can "Include" another, giving rise to a tree DF files. This compact hierarchy concentrates in one place information that is otherwise scattered, and is undoubtedly an aid to perceiving structure.

A system model written in SML is an extremely detailed applicative program

describing how the modules making up the system are to be bound. It augments the import and export lists of the Cedar source program, specifying precisely which version of each module is to be used in each situation. Module types appear in the model, and are checked for consistency. Models are updated automatically when modules are edited, so as accurately to reflect the interconnections specified in the source program. A *system modeller* executes SML models, thereby constructing the specified systems.

SML models can be structured hierarchically, and accordingly they support hierarchical organisation of systems. Like MIL 75, they specify interactions explicitly, though in a rather different way. SML emphasises version control, but a system model serves also as a means of documenting structure. The fact that models are updated automatically when modules are changed, however, implies that SML is not used to enforce a predetermined structure. It also provides no support for layered structures.

1.4.5. PIE

PIE is a personal database system, used for organising and manipulating multiple versions of programs, documents or other structured information [Bobrow-Goldstein 80, Goldstein-Bobrow 80a, Goldstein-Bobrow 80b, Goldstein-Bobrow 80c, Goldstein 81]. Its best-known use is as a programming environment and version control mechanism for Smalltalk [Ingalls 78], and it is in this context that it will be described here.

A Smalltalk program is generally represented as a network; PIE extends this model to include layers for handling version control. A *layer* encapsulates a collection of related changes made to the network for some purpose. A *context* is a sequence of layers. When a context is *installed*, the changes specified in its layers take effect; changes in layers occurring early in the sequence *dominate* conflicting changes in layers occurring later in the sequence. This is an elegant

mechanism for isolating and categorising changes in such a way that any desired version can be specified and installed. In addition, nodes representing layers and contexts are themselves added to the network and can have documentation associated with them, making the system self-contained and encouraging the user to provide full and structured documentation.

Another interesting feature of PIE is the *perspective* mechanism: a number of special objects called perspectives can be attached to a node in the network, each reflecting a different view of the entity represented by that node and providing operations, called *methods*, for manipulating that view. This mechanism therefore allows multiple views of objects to be specified and manipulated. It is independent of layers, however, which are used only for version control.

PIE is of interest here because of the perspective mechanism and its use of layers. It is not really a structuring mechanism, however, as no specification of structure is involved: the network PIE deals with is the program itself. This means that any display of structure provided by PIE must be derived from the program, and that PIE is not used for enforcement of structure.

1.4.6. Structured Analysis

Structured Analysis (SA) is a language for communicating ideas [Ross-Schoman 77, Ross 77, Dickover-McGowan-Ross 77]. It provides a structural framework within which anything can be expressed in any language, and hence is an extremely general structuring mechanism. Its primary intended use is as a blueprint language for the specification of computer software, and it is the basis of Structured Analysis and Design Technique (SADT[8]), SofTech's proprietary systems analysis and design methodology [Ross-Schoman 77, Dickover-McGowan-Ross 77].

---

[8]SADT is a trademark of SofTech, Inc.

An SA *model* consists of an hierarchy of descriptions, each of which is a single-page diagram. Diagrams are constructed from some 40 symbols, chiefly boxes, arrows and annotations. A single model specifies a particular aspect of a system (called *viewpoint*) at a particular level of abstraction (called *vantage point*) [Ross-Schoman 77]. A complete specification of a system consists of multiple models covering a number of viewpoints and vantage points, as determined by the purpose of the specification. The organisation of an SA specification is more complex than any of the program organisations considered in Section 1.1.1. However, if only a single viewpoint is used, the specification has a layered organisation in which each layer is an hierarchical model describing the system from a particular vantage point.

The primary purpose of SA is to serve as a design tool and a language for human communication, rather than a structuring mechanism for computer programs. Its diagrammatic nature serves this end well, but makes it unsuitable for automatic processing as it stands. It therefore has no mechanism, other than discipline, for enforcing the structures it specifies, and is unsuitable as a representation of structure for use by software.

1.4.7. Summary

Though many of the structuring mechanisms described above satisfy some of the requirements listed in Section 1.3, none satisfies all the requirements. The most important shortcoming is that none of the mechanisms providing automatic enforcement of structure can handle layered, object-oriented programs adequately. The grid mechanism was specifically designed to do so.

1.5. The Grid Mechanism

The grid mechanism was designed to meet all the requirements discussed in Section 1.3. It derives its name from the fact that it arranges units on a two-dimensional grid, one dimension corresponding to layer and the other to object.[9] It thus provides explicit support for layered, object-oriented programs, clearly identifying the layers and objects and the correspondence between units in different layers. It uses four techniques, called *factorisation*, *clustering*, *deviation* and *approximation*, to help manage complexity, simplify structure description, and identify similarities of structure between layers. These techniques are described in detail in Chapter 2.

In addition to satisfying its fundamental requirements, the grid mechanism also has the following properties:[10]

- Multiple grids can be used to specify the structure of a single program from multiple points of view. This is analogous to multiple indexes into a single database.
- Multiple relationships between program units can be specified, either within a single grid, or by means of separate grids.
- The grid can handle multiple, alternative implementations of a single specification. Such alternative implementations do not occur frequently, but are useful in some important special cases.
- The grid can be specified as an abstract data type, which then serves as a uniform interface to all software associated with it, such as editors, browsers and compilers.
- The grid is completely programming-language independent, and can even be used to specify the structure of entities other than programs. A potential problem with language-independent mechanisms is that the structural information they specify might be inaccessible to the language system (compiler, debugger, etc.). This problem is solved in the case of the grid mechanism by providing an interface through which communication with a language system can take place.

---

[9] There is no reason why the grid mechanism could not be used in other cases of two simultaneous categorisations, or be extended to handle more than two.

[10] The prototype implementation of the grid does not yet have all these properties; the design of the grid, and of the implementation, ensures that they can be added gracefully.

- The syntax of the grid can be tailored to that of any language, allowing it to be integrated gracefully with any existing programming language.
- A grid is textually separate from the program whose structure it specifies. This has several advantages:
    - o The structure specification can be perused separately from the program, and used as an index into the program.
    - o Structural information is concentrated in one place, rather than being scattered throughout the program.
- A grid describing program structure can be wholly or partially constructed before the program is written, and then used to control program development.
- A grid can be constructed from an existing program, either automatically or with guidance from the user. This process often highlights errors or structural peculiarities previously unnoticed by the author of the program.
- The grid provides a convenient, structured repository for information about a program, such as documentation. This facility can also be used by source management, project management and version control systems, and the like.

Though the requirements for the grid mechanism were motivated by the nature of Fable and the Fable environment, they are not peculiar to Fable; nor are the additional properties listed above. Large, layered, object-oriented programs arise in many other contexts also, as described in Section 2.1, and a structuring mechanism suited to specifying, documenting, enforcing and representing their structure would be of general use. Since the grid mechanism is completely language independent, it is suitable for use in other contexts. Henceforth it is described as a general structuring mechanism, without further reference to Fable.

1.6. A Simple Example

This section introduces the grid mechanism by means of a small, yet important, example; details of the grid are given in subsequent chapters. The example program is described, and its structure is specified using first nesting, then visibility control lists, and finally the grid.

The example is based on a discussion by Woitok of alternative implementations of the abstract data type *list* [Woitok 83]. Consider a program consisting of the following objects:

- An abstract data type, *C*, implementing complex numbers. It provides operations *plus* and *minus*, among others, and is implemented as a pair, (x, y), of reals.
- An abstract data type, *L*, consisting of lists of elements of type *C*. It provides operations *head* and *tail*, among others.
- Two user procedures, *CU* and *LU*; *CU* uses *C*, and *LU* uses *L*.

The nature of the program is such that the same complex number is never placed on more than one list.

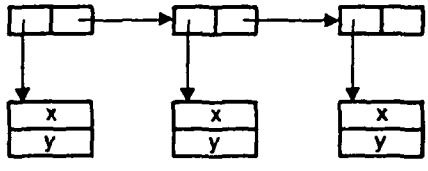


Figure 1-1: Cons Cell Implementation

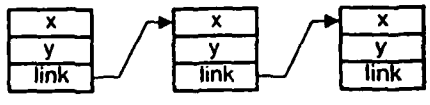


Figure 1-2: Link Field Implementation

Suppose a linked implementation of *L* is to be used. If one is concerned about data abstraction and information hiding, the traditional way of doing this is by means of cons cells, as illustrated in Fig. 1-1. An alternative implementation, which is more appealing in many situations for reasons discussed by Woitok, involves incorporating a link field into the implementation of type *C*, as illustrated in Fig. 1-2. However, this approach violates the abstract data type discipline, because part of the implementation of *C* is affected by and used by *L*.

Woitok solves this problem by introducing *incremental records*: records whose fields are declared and used in disjoint scopes, but are implemented contiguously. Multiple views and the grid mechanism can be used to achieve a similar effect, without the introduction of incremental records into the programming language.

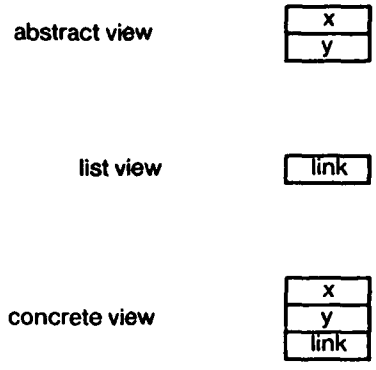


Figure 1-3: The Three Views of *C*

The key to the grid solution is the identification of three *views* of *C*, illustrated in Fig. 1-3:

- The *concrete* view describes the actual implementation of *C* as a record, with *link* field included. As in the case of classical abstract data types, this view is hidden from all other objects.
- The *list* view consists just of the operations *get-link* and *set-link* for manipulating the *link* field, and *null* and *equal* on objects of type *C*. They are implemented in terms of the concrete view in the obvious way. This view is available only to *L*.
- The *abstract* view consists of the operations defined on *C*, with no reference to lists. These operations are implemented in terms of the concrete view, as in the case of classical abstract data types. This view is generally available to users.

The other objects are also arranged within this view framework, and appear in the abstract view. The abstract view of *L* consists of the list operations, which are implemented in terms of the list view of *C*; in other words, the

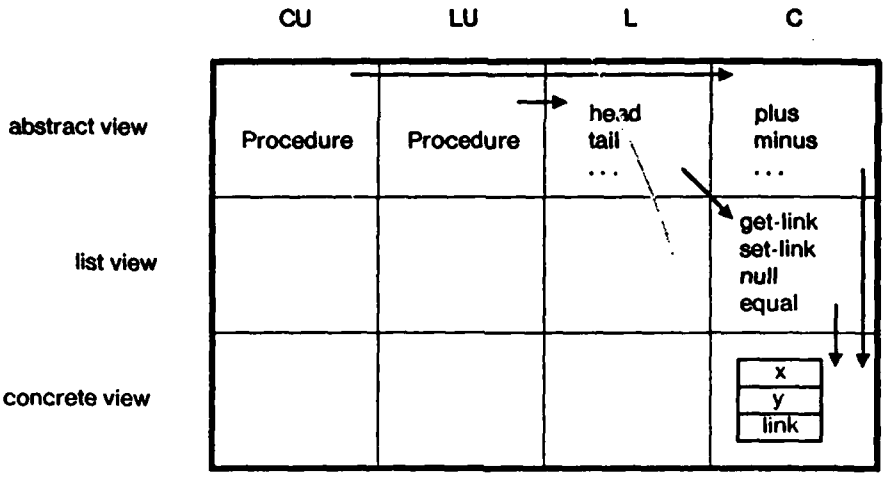


Figure 1-4: Overall Structure

representation of type *L* is the list view of type *C*.[11] The users *CU* and *LU* are also placed in the abstract view, and can interact only with other units in the abstract view. Fig. 1-4 presents the overall picture. The arrows in the figure represent permitted interactions, or references.

This example program is an extremely simple layered, object-oriented program. To specify its structure adequately, in accordance with the requirements listed in Section 1.3, a structuring mechanism must be able to:

- Specify the dual categorisation of units according to object and view illustrated in Fig. 1-4.
- Specify the permitted interactions accurately. The following two restrictions that ensure proper information hiding are particularly important:
  - The concrete view of *C* can be used only by other views of *C*.
  - The list view of *C* can be used only by *L*.

The remainder of this section examines the extent to which three structuring mechanisms achieve these aims: nesting, visibility control lists and the grid mechanism.

---

[11] An alternative way of handling *L* is to define a concrete view as well, equate it to the list view of *C*, and specify that the abstract view of *L* is implemented in terms of the concrete view of *L*. This is an elegant approach, but slightly more complex.

1.6.1. Nesting

For the purposes of this example, assume the availability of Ada-like nested modules: each module consists of a *visible part* and a *hidden part*, and either part can contain nested submodules [ANSI 83]. In this context, organisation is specified by grouping units into modules, and information hiding is accomplished by placing units to be hidden in the hidden parts of their modules.

In this simple example, the dual categorisation of units according to object and view can specified quite well by associating a module with each object, and a submodule with each view of that object. The fact that only abstract views are generally available suggests placing abstract view submodules in the visible parts of their parent modules, and all other views in the hidden parts. The result is shown in Fig. 1-5(a). All the modules are assumed to be at the top level, and hence available to one another. There are two problems with this solution:

- All abstract views can potentially interact with all other abstract views, whereas Fig. 1-4 specified only two specific interactions between abstract views. This problem is unfortunate, but not critical.[12]
- The list view of C is not accessible to L. This problem is critical, for such access is essential.

The list view of C can be made accessible to L in one of two ways:

- Placing it in the visible part of module C, as shown if Fig. 1-5(b). This also makes it visible to all users of the abstract view of C, a grave violation of information hiding.
- Making module L a submodule of the visible part of C, as shown in Fig. 1-5(c). This makes the concrete view of C accessible to L, another grave violation of information hiding. Also, this organisation is somewhat inferior, as objects C and L are no longer well separated.

No nested solution is possible that does not involve one or other of the above information hiding violations.

---

[12] Interactions can, of course, be restricted by the use of visibility-control lists, if they are available in addition to nesting. A solution combining nesting and visibility control lists is presented in Section 1.6.2.

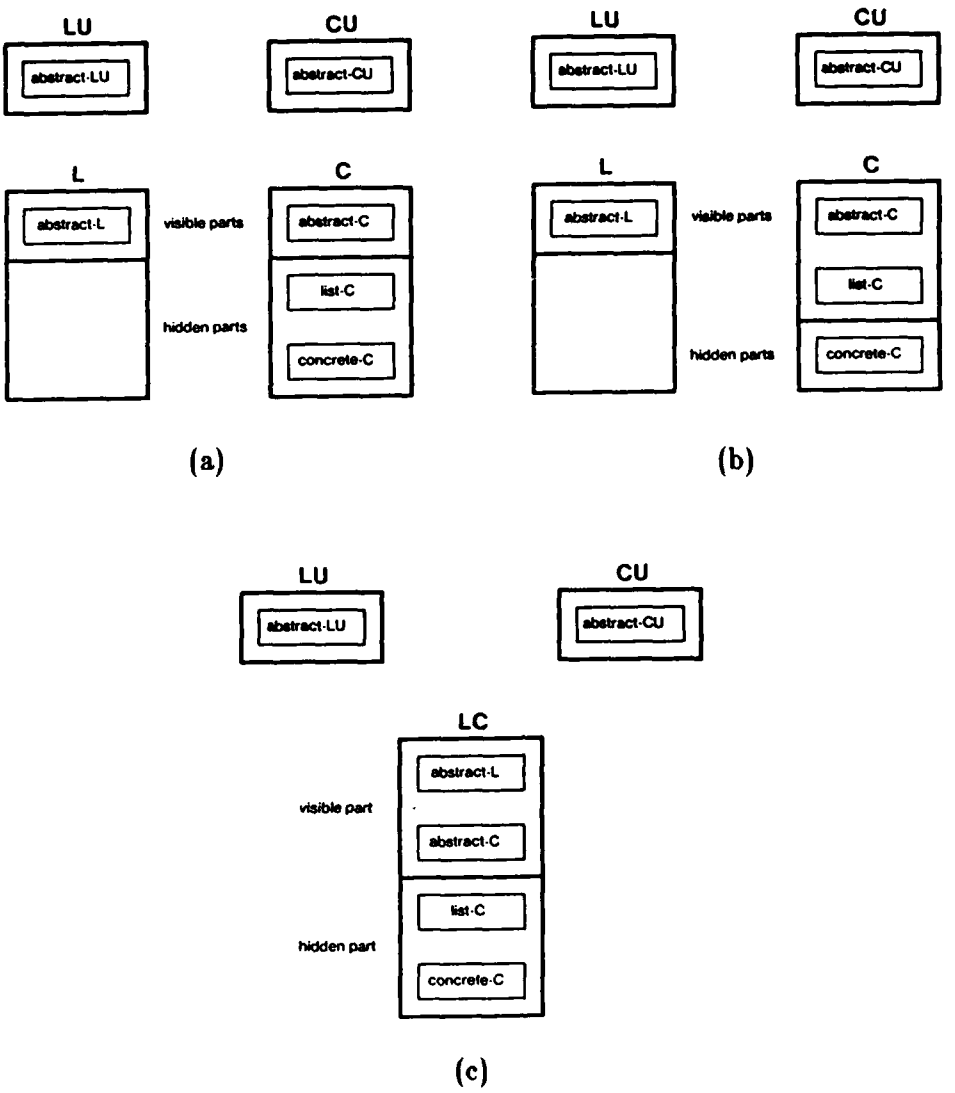


Figure 1-5: Three Nested Solutions

In summary, nesting is unable to specify the structure of this example program. Nesting supports information hiding adequately in the common and important case where the hidden unit is accessible to exactly one higher-level unit, its parent; nesting always fails when the hidden unit must be accessible to two or more specific higher-level units.

```
module abstract-CU;
    imports abstract-C;
    . . .
end;

module abstract-LU;
    imports abstract-L;
    . . .
end;

module abstract-L;
    imports list-C;
    . . .
end;

module abstract-C;
    imports concrete-C;
    . . .
end;

module list-C;
    imports concrete-C;
    . . .
end;

module concrete-C;

    . . .
end;
```

Figure 1-6: A Solution with Import Lists

```
module abstract-CU;

    . . .
end;

module abstract-LU;

    . . .
end;

module abstract-L;
    exported to abstract-LU;
    . . .
end;

module abstract-C;
    exported to abstract-CU;
    . . .
end;

module list-C;
    exported to abstract-L;
    . . .
end;

module concrete-C;
    exported to abstract-C, list-C;
    . . .
end;
```

Figure 1-7: A Solution with Export Lists

1.6.2. Visibility Control Lists

Visibility control lists do not specify organisation at all, so they cannot make the dual categorisation explicit. They can, however, always specify interactions with complete accuracy. Figs. 1-6 and 1-7 show import and export lists, respectively, that specify the permitted interactions shown in Fig. 1-4. In the case of a small program such as this, visibility control lists are reasonably adequate; there are few enough units and interactions that one can gain an

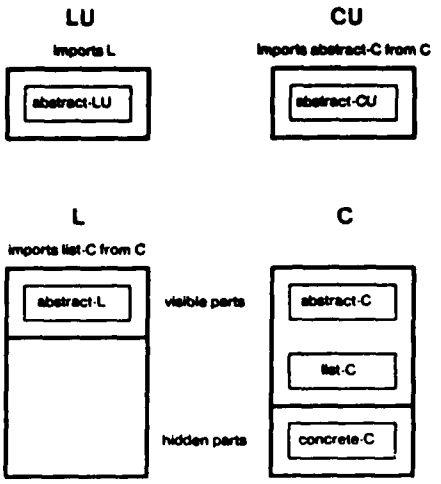


Figure 1-8: A Combined Solution

appreciation of the overall structure of the program by examining the lists. This is not true of large programs.

Perhaps the best way of structuring the example program using traditional mechanisms is by means of a combination of nesting and detailed visibility control lists. Fig. 1-8 shows a combination of the nested modular organisation of Fig. 1-5(b) with detailed import lists that allow only *L* to access the list view of *C*. Though the modular organisation was patterned after Ada, this solution cannot be programmed in Ada because *with* clauses cannot mention submodules.

1.6.3. The Grid Mechanism

The grid mechanism specifies program organisation by arranging units on a two-dimensional grid, called the *matrix*. The rows of the matrix correspond to views, and the columns to objects. Fig. 1-4, without the arrows, is thus an accurate illustration of the grid matrix.

Instead of specifying interactions between individual units in the matrix, as was done in Fig. 1-4, the grid mechanism specifies interactions between objects

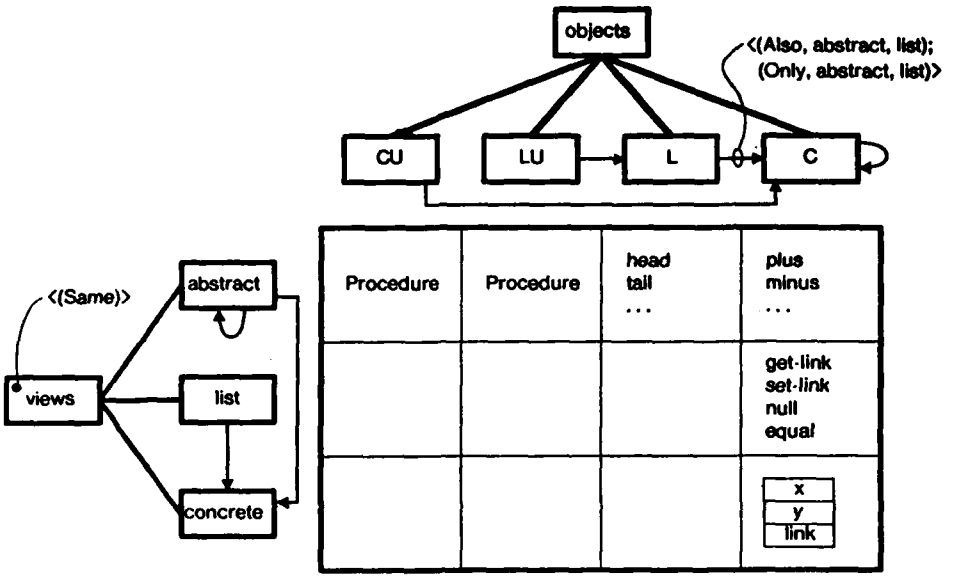


Figure 1-9: A Grid Solution

in an *object directory*, and interactions between views in a separate *view directory*. The complete grid is shown in Fig. 1-9.

The directories are hierarchies, allowing for hierarchical classification and decomposition; in this simple example, they contain only one level below the roots. The arrows in the directories specify permitted interactions; they are always between siblings in the hierarchy, and hence are called *sibling interactions*. The annotations attached to directory nodes and arrows are called *qualifiers*, and specify restrictions and/or additional permitted interactions.[13] In general, an interaction between two units is considered permissible if and only if:

- The arrows and qualifiers in the object directory specify an interaction between the objects they describe, *and*
- The arrows and qualifiers in the view directory specify an interaction between their views.

However, an *Also* qualifier in one directory specifies an interaction that is always permissible, irrespective of the arrows and qualifiers in the other directory.

[13]Detailed discussion of the grid in succeeding chapters should clarify the need for both sibling interactions and qualifiers as means of specifying interactions.

Consider the view directory first. The arrows specify that units in the abstract view can use units in both the abstract view and the concrete view, and that units in the list view can use units in the concrete view. These arrows alone allow free access to the concrete view, which would be a violation of information hiding. The *Same* qualifier restricts such access to units describing the same object, thereby capturing the essence of the information hiding requirements.[14] Note that no access to the list view is provided in the view directory.

The arrows in the object directory correspond to the horizontal components of the arrows in Fig. 1-4, and most of them specify interactions between objects in the obvious way. The qualified arrow from L to C is of special interest, and is interpreted as follows:

- The arrow itself specifies that L uses C, as expected.
- The *Also* qualifier specifies that the abstract view of L uses the list view of C, despite the arrows and the *Same* qualifier in the view directory.
- The *Only* qualifier specifies that the interaction between the abstract view of L and the list view of C is the only permissible interaction between L and C. It explicitly excludes the interaction between the abstract view of L and the abstract view of C, which would otherwise be considered permissible; it also excludes the interaction between the abstract view of L and the concrete view of C, but this has already been excluded by the *Same* qualifier in the view directory.

This grid specification illustrates a number of important properties of the grid mechanism:

- The dual categorisation of units according to object and view is specified explicitly by the matrix.
- By looking at the object directory in isolation, one can get a general picture of how the objects interact, without being concerned with details of the views.
- By looking at the view directory in isolation, one can get a general

---

[14] More precisely, the *Same* qualifier states that an interaction between two units is permissible only if they belong to the same object and/or the same view.

picture of how the views interact, without being concerned about the particular objects involved. In fact, the view directory specifies a particular information hiding discipline that might be useful in other situations.[15]

- The arrows tend to specify "ordinary" interactions, whereas the qualifiers specify restrictions or "exceptional" interactions. One can thus get an overview of structure by looking just at the arrows, and can concentrate on irregularities or peculiarities by looking just at the qualifiers.
- Violations of information hiding can be specified when needed, but they are always highlighted by qualifiers.

Though this example is tiny, it is an interesting illustration of the capabilities of the grid. It is not, however, a convincing demonstration of the superiority of the grid over visibility control lists, which are adequate for small programs, but do not scale well. Chapter 5 describes the use of the grid to specify the structure of two real, large programs (12000 and 29000 lines), and shows that the grid is indeed effective at specifying and documenting the structure of large programs as well.

---

[15] For example, the view directory specifying the standard abstract data type discipline is similar to this one, except that the list view is absent and the concrete view is allowed to access the abstract view without the *Same* restriction.

Chapter 2
Layered Graph Structures

The grid mechanism was designed to specify the structure of layered, object-oriented programs, introduced briefly in Chapter 1. Section 2.1 describes such programs in detail, and introduces an example program that is used as a running example throughout this thesis. Section 2.2 introduces layered graphs as an appropriate model of the structure of layered, object-oriented programs. Section 2.3 outlines the techniques developed for specifying complex layered graph structures, and the remaining sections describe the techniques in detail.

2.1. Layered, Object-Oriented Programs

A layered, object-oriented program describes and manipulates a number of *objects*, some or all of which are described from *multiple points of view*. Each layer in such a program corresponds to a viewpoint: the units in one layer describe different objects, but from a single, consistent point of view. Units in different layers can describe a single object, but from different points of view. Multiple views of objects arise whenever multiple levels of abstraction are used: each level of abstraction corresponds to a view. Multiple views also arise when a shared object is used in different ways by different users: each user has its own view of the shared object.

> **Example 2-1:** A simple example of a system in which multiple views of objects arise as a result of levels of abstraction is an electronic mail system. Objects in such a system include messages, buffers and name tables. At least three views of a message can be discern[illegible]:

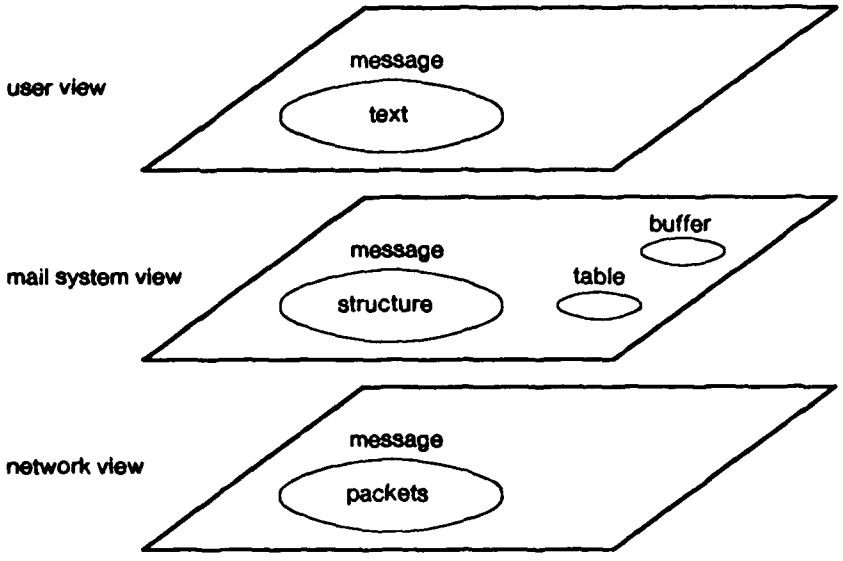


Figure 2-1: Multiple Views of a Mail Message

- *User view*: a message is a piece of text.
- *Mail system view*: a message might be a structure consisting of such fields as source, routing information, time stamp and text.
- *Network view*: a message might be a collection of packets.

This situation is illustrated in Fig. 2.1. The three layers shown correspond to the three views identified above. A view of the message object appears in each layer. Buffers and name tables, on the other hand, appear only in the mail system view, as they are of no concern either to the user or the network.

A more complex example, based on a shared object used in different ways by different users, appears in Section 2.1.1.

Layered, object-oriented programs as used in this thesis incorporate the important notion of separating *definition* from *implementation*. This notion was introduced by the data abstraction languages CLU [Liskov, *et al.* 77, Liskov, *et al.* 81] and Alphard [Wulf-London-Shaw 76], and since then has been used in many modern programming languages such as Ada [ANSI 83], Mesa [Mitchell-Maybury-Sweet 79] and Modula-2 [Wirth 83]. Each unit of a layered, object-

oriented program describes a particular view of a particular object, and is assumed to consist of a *definition subunit* and one or more *implementation subunits*.[16] The definition subunit defines the view of the object by specifying the facilities, such as operations, that it makes available. Each implementation subunit specifies how these facilities are implemented, either in terms of other views of the same object, or perhaps in terms of other objects. Multiple implementation subunits are allowed, to facilitate the specification of multiple, alternative implementations of the same view. Though not widely used, this facility is useful in some important cases.

> **Example 2-2:** In the case of the mail system example, the user view of messages is implemented in terms of the mail system view, and the mail system view in terms of the network view. Consider the relationship between the user and mail system views:
>
> - The definition subunit of the user view of messages will define them as text, and specify the operations that can be performed on them by the user, such as sending and receiving.
> - The definition subunit of the mail system view of messages will define them as structures, and specify the operations that can be performed on them by the mail system, such as constructing and enqueuing them.
> - The implementation subunit of the user view of messages will specify how each user operation is implemented in terms of the mail system operations on messages, and perhaps on other objects that form part of the mail system. For example, a user's message might be sent by constructing a suitable message structure from the text and enqueuing it on an appropriate queue.

Multiple views of objects are commonly encountered in many fields. An automobile has an owner's manual and a mechanic's manual, describing it from

---

[16] Definitions are also commonly known as *specifications* or *interfaces*, and implementations as *bodies* or *program modules*. An alternative approach is to consider definitions and implementations to be full-fledged units, rather than subunits, and to deal with two categories of units. This approach has some advantages, but is less convenient for the purposes of the grid mechanism. This issue is discussed further in Chapter 6.

different points of view. The same is true of most appliances. Engineering blueprints describe and illustrate a single object from many points of view, such as side-view and cross-section. Many database systems allow the data they store to be viewed in different ways, and the database itself can be viewed at the physical, conceptual and view (or subscheme) levels [Ullman 82]. The PIE system, described in Section 1.4.5, has an explicit mechanism, called *perspectives*, for specifying multiple views of objects [Goldstein-Bobrow 80a], and this mechanism has been carried forward into the Loops language [Bobrow-Stefik 83]. Structured Analysis, described in Section 1.4.6, can be used as a blueprint language for software, and allows software objects to be described from multiple viewpoints and vantage points [Ross 77]. In any class hierarchy [Ingalls 78, Bobrow-Stefik 83], each ancestor of an object is a view of that object at some level of abstraction. In the Fable language for specifying integrated circuit manufacturing processes, both processes and the equipment that performs them are described at various levels of abstraction, giving rise to multiple views of each process and each piece of equipment [Ossher-Reid 83].

Some programming languages provide facilities for specifying multiple views of objects explicitly. Mesa is a good example: multiple *interface modules* associated with a single *program module* can be used to specify multiple views of a single object. This usage does not seem to be common in practice, however. At Xerox Systems Development Division (SDD), for example, programmers tend to write a single interface for each module, incorporating everything that any user might need and often including default parameters that can be ignored when not needed [Elliott 84]. The chief technical reason for this is probably that inadequate programming environment support is provided for multiple views. Many of the views of an object are often very similar, and in the absence of tools to assist with their specification, a great deal of source-code duplication is required. If good programming environment support were provided, this usage might become more popular, though there would still be cultural issues to overcome: Mesa programmers, at least within Xerox SDD, tend to use carefully developed and well-entrenched standard programming practices [Elliott 84].

Thus, despite the fact that multiple views of objects occur so naturally and frequently, they are seldom made explicit within computer programs. This does not argue against the layered, object-oriented model, however. The model itself has much intuitive appeal, and I believe it can result in clean and well-structured programs. If good programming language environment support were provided for it, such as by a complete environment based on the grid mechanism, it might gain sufficient users that its merits could be judged from practical experience.

2.1.1. Example: A Simple Shared Database

This section describes a layered, object-oriented program that is used as a running example throughout this thesis. It is a simple case of a shared resource being used in different ways by different users.

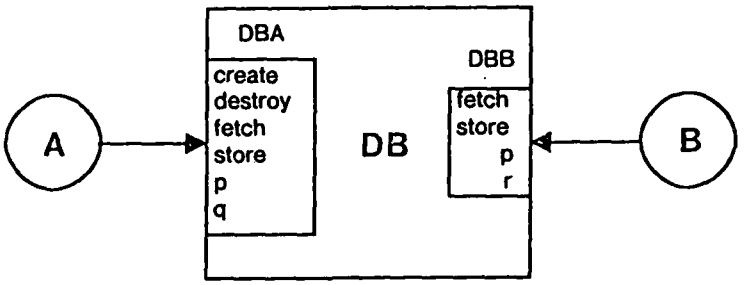


Figure 2-2: Two Packages Sharing a Database

Example 2-3: The shared database program consists of ten *objects*:

- The central object is an extremely simple, in-core database, *db*, consisting of a network of named, attributed nodes. It provides operations *create*, *destroy*, *fetch* and *store*, and supports attributes *p*, *q* and *r*.
- Two objects, *a* and *b*, use *db*; *a* uses all the operations provided by *db*, and manipulates attributes *p* and *q*, whereas *b* uses the *fetch* and *store* operations only, and manipulates attributes *p* and *r*.

- Two objects, *hash* and *random*, are used to implement *db*.
- Objects *values*, *alists*, *lists*, *pairs* and *strings* export data types for general use and are referred to as *utilities*.

The interactions between *a*, *b* and *db* are illustrated in Fig. 2-2.

Three different *views* of *db* are thus present in this example:

- The complete view, *DB*, containing details of all the operations and attributes.
- *a*'s view, *DBA*, containing just those aspects of *db* that are visible to *a*.
- *b*'s view, *DBB*, containing just those aspects of *db* that are visible to *b*.

Three *units* describe these views, each of which is an Ada package consisting of a package specification and a single package body. The specifications are shown in Fig. 2-3. The bodies of *DBA* and *DBB* implement their views in terms of the complete view, *DB*, and the body of *DB* implements the complete view in terms of the objects *hash*, *values*, *alists* and *strings*. Each of the other objects is described by means of a single unit: the names of these units are *A*, *B*, *H*, *R*, *V*, *AL*, *L*, *P* and *S*, corresponding to objects *a*, *b*, *hash*, *random*, *values*, *alists*, *lists*, *pairs* and *strings*, respectively. These units are not shown, as their details are of little concern here.

Two kinds of interactions between units are of importance in this example, and indeed in most programs in which definitions and implementations are distinguished. Each is characterised by a *relation*. The relation *DefUses* relates a unit to all the units used by the definition subunit of that unit. Similarly, the relation *ImpUses* relates a unit to all units used by the implementation subunit of that unit. These relations are illustrated graphically in Figs. 2-4 and 2-5, primarily to show their complexity and the inadequacy of direct graphical representations. The specification techniques on which the grid mechanism is based are applied to these relations in subsequent examples.

```
with V, S; use V, S;
package DB is
  type attribute is (p, q, r);
  procedure create(node: string);
  procedure destroy(node: string);
  function fetch(node: string; attr: attribute) returns value;
  procedure store(node: string; attr: attribute; val: value);
end DB;

with V, S; use V, S;
package DBA is
  type attribute is (p, q);
  procedure create(node: string);
  procedure destroy(node: string);
  function fetch(node: string; attr: attribute) returns value;
  procedure store(node: string; attr: attribute; val: value);
end DBA;

with V, S; use V, S;
package DBB is
  type attribute is (p, r);
  function fetch(node: string; attr: attribute) returns value;
  procedure store(node: string; attr: attribute; val: value);
end DBB;
```

Figure 2-3: Specifications of *DB*, *DBA* and *DBB*

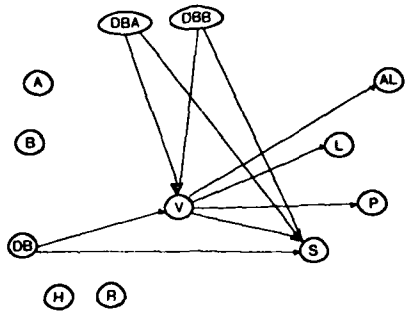


Figure 2-4: The *DefUses* Relation

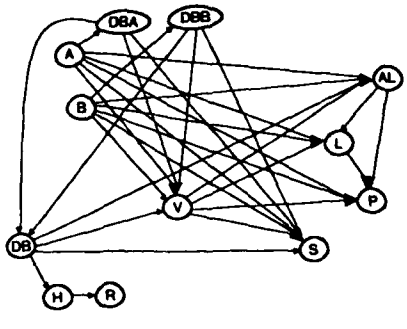


Figure 2-5: The *ImpUses* Relation

2.2. Layered Graphs

Graphs are commonly used to model and illustrate program structure, as in Figs. 2-4 and 2-5. The nodes of a graph represent program units, and the edges represent interactions between units. Since multiple kinds of interactions between units can occur, and each is characterised by a separate relation, the following definition of a graph is appropriate in the current context:

> **Definition 2-1:** A *graph* is a pair (N, R), where N is a set of nodes, and R is a set of relations on N specifying interactions between the nodes.

Throughout this thesis, all relations are assumed to be binary.

A particularly natural and convenient model for the structure of a layered, object-oriented program is a special kind of graph called a *layered graph*. A layered graph consists of a number of horizontal layers, each of which is a graph. Each node in the layered graph represents the unique unit describing a specific object from a specific point of view. Each layer corresponds to a viewpoint, and is called a *view slice*, or simply *view*: the nodes in a view describe different objects, but from the same point of view. All nodes in the various views that describe a single object can be arranged on a vertical plane orthogonal to the views. Each such plane corresponds to an object, and is called an *object slice*: the nodes in an object slice describe the same object, but from different points of view. Edges in the graph represent interactions between units, and can cross object slice and view boundaries arbitrarily.

> **Example 2-4:** A layered graph that models the structure of the shared database program is shown in Fig. 2-6. There are four views, called *a*, *b*, *db* and *u*. View *a* specifies the program structure from *A*'s point of view: it contains nodes representing *A* and all units used by *A*, and shows how they interact. Similarly, views *b* and *db* specify the program structure from *B* and *DB*'s points of view. The fourth view,

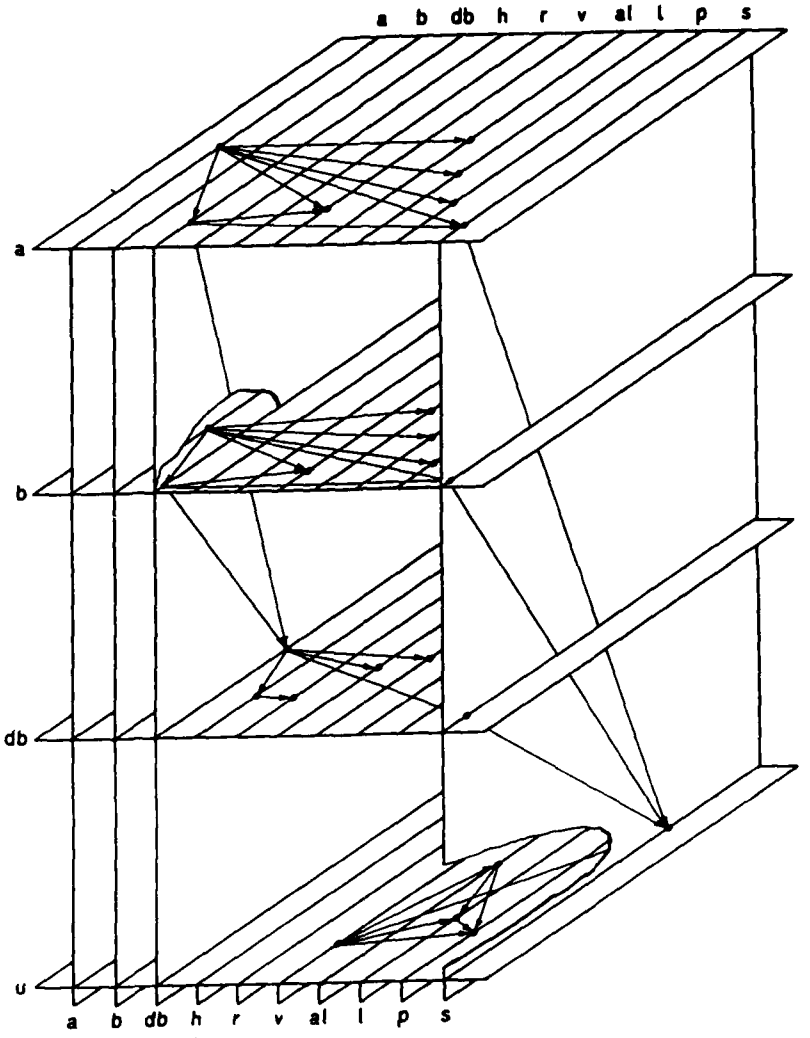


Figure 2-6: Layered Graph

u, contains nodes representing the utilities, and shows how they interact. There are ten object slices, called *a*, *b*, *db*, *h*, *r*, *v*, *al*, *l*, *p* and *s*, corresponding to the objects *a*, *b*, *db*, *hash*, *random*, *values*, *alists*, *lists*, *pairs* and *strings*, respectively. All nodes and all views are shown in full, but only four object slices, to keep the diagram intelligible. The omitted object slices *h* and *r* contain just single nodes, and the omitted object slices *v*, *al*, *l* and *p* are similar to object slice *s*. The arrows represent pairs in the relation *ImpUses*; for simplicity, this single relation is used in all examples appearing in this chapter.

As expected from the description of the example, three different views of the database are present in the graph. Less expected, perhaps, are the multiple views of the utilities. Consider the utility ***strings***, which is typical. Its object slice, *s*, contains four nodes, corresponding to four units called *SA*, *SB*, *SDB* and *S*. Each unit describes a distinct view of ***strings***. Since all these views are in fact identical, they were not distinguished in the original description of the program; they are introduced here to ensure that each layer in the graph presents a complete picture of the program structure from the appropriate point of view: since *A*, for example, uses ***strings***, a view of ***strings*** must appear in layer *a*.[17] Each of *SA*, *SB* and *SDB* consists of the same definition as *S*, and a trivial implementation implementing the definition in terms of *S*.[18] These implementations account for the arrows lying within object slice *s* in the figure.

In this example, each interaction crosses either an object slice boundary or a view boundary, but never both. This is quite common in well-structured layered graphs, but is not necessarily the case. In general, interactions can be arbitrary.[19]

A layered graph as used in this context is defined as follows:

Definition 2-2: A *layered graph* is a triple (G, V, R), where G and V are partitions of a set, N, of nodes, and R is a set of relations on N specifying interactions between the nodes. The members of G are

---

[17] An alternative approach is also possible, in which only a single view of each utility, the *u* view, is present, and users in all views use this single view. No additional units are introduced in this case, but a view can no longer be said to contain a complete picture of the program structure from a particular point of view. This approach was followed in specifying the structure of Scribe, as described in Section 5.3

[18] These identical views and their trivial implementations would not have to be specified by source-code duplication in a sophisticated programming environment; they could merely be declared to exist. This provides the conceptual clarity of having a separate view for each user, without introducing the many difficulties associated with maintaining multiple copies.

[19] For example, if the alternative approach to the handling of the utilities is used, all arrows from user nodes to utilities lead down to the *u* view, crossing both object slice and view boundaries.

called *object slices*, and the members of V are called *view slices*, or simply *views*.[20] The partitions G and V must have the property that the intersection of any object slice and any view contains at most one node.[21]

An important consequence of the restriction on G and V is that any node can be uniquely identified by specifying its object slice and view. The syntax g/v is used to denote the node in object slice g and view v.

The only formal difference between the layered graph (G, V, R) and the graph (N, R) over the same set of nodes is that the object slices and views are identified explicitly. This is a crucial difference, however, as specification of grouping is one of the most important aspects of documenting structure.

2.3. Techniques for Specifying Layered Graph Structures

The interactions between nodes in a large layered graph can be so complex that no direct representation of them would make the structure readily visible. This Section introduces techniques for simplifying the specification of layered graph structures. The techniques are based on the following principles:

- *Abstraction.* It is easier to understand a complex system if it is presented at multiple levels of abstraction. A reader can then begin at the highest level and proceed to more detailed levels as and when desired.
- *Grouping.* It is easier to understand a complex system if coherent collections of parts are identified and used in describing it.

---

[20] The symbol "G" derives from the original use of the term "group" for "object slice". Since "O" is already an overloaded mathematical symbol, retaining "G" seems as good as any alternative.

[21] The names *object slices* and *views* reflect the interpretation placed on these partitions by the grid mechanism. This definition, as well as the techniques for specifying layered graph structures described later, in no way depend on this interpretation. The techniques, and the grid mechanism itself, are therefore valid in dealing with any graph having two separate partitions of nodes with the required property. They could also be trivially extended to the case of more than two partitions.

- *Deviation.* It is easier to understand a complex system if it is specified as a similar system that is simpler or more familiar, together with details of how they differ. The utility of this principle depends on how clearly and concisely the differences can be specified.
- *Approximation.* It is easier to understand a complex system if all unnecessary detail is omitted.

Abstraction and grouping are widely used in computer science, whereas deviation and approximation are relatively untried in this area, though common in everyday life.[22] An interesting application of the principle of deviation in computer science is the *define* command in Scribe [Reid 80, Reid-Walker 80]. This command allows a new *environment* to be defined as equivalent to an existing one, except for explicitly specified differences; the user need not even understand all the details of the existing environment in order to use it.

The simplification techniques underlying the grid mechanism result from applying the above principles to the specification of layered graph structures. Let $L = (G, V, R)$ be a layered graph, as defined in Section 2.2, where $R = \{ R1, R2, \ldots, Rk \}$ is a set of relations on the nodes making up partitions G and V. The techniques are outlined here, and described in detail in subsequent sections:

- *Factorisation.* Interactions between nodes are replaced by interactions between object slices and interactions between views. The result is called the *pure factored form* of L,

  $$Lpf = (GG, VG)$$

  where GG is the *object graph* specifying interactions between object slices, and VG is the *view graph* specifying interactions between views. Lpf is an abstraction of L that contains considerably less detail, and is therefore not, in general, an accurate specification of L; in special cases where it is, the layered graph is called *regular*.
- *Clustering.* This technique is applied to both the object graph and the view graph of the factored form of a layered graph. Nodes representing object slices or views are grouped into *clusters*, and

---

[22] Consider descriptions such as "the table-top is rectangular, but with rounded corners", or "the flower bed surrounds the lawn, except for a gap at the gate".

interactions between nodes in different clusters are replaced by interactions between the clusters themselves. The technique is applied recursively until all clusters are small and simple. The result is called the *pure clustered form* of L,

$$Lpfc = (GGpc, VGpc)$$

where $GGpc$ and $VGpc$ are the results of applying the process of clustering to GG and VG respectively. $Lpfc$ is an abstraction of Lpf that contains considerably less detail, and is therefore not, in general, an accurate specification of Lpf; in special cases where it is, the clustering is called *uniform*.

- *Deviation*. The techniques of factorisation and clustering allow a regular, uniformly clustered graph to be specified accurately, clearly and concisely in pure clustered form. The technique of deviation extends the advantages of such specification to arbitrary layered graphs. It allows an arbitrary layered graph, L, to be specified as a pure clustered graph, $Lpfc$, together with *qualifiers* specifying exactly how L differs, or *deviates*, from $Lpfc$. Thus L is specified in *qualified clustered form* as the triple

  $$Lfcq = (GGpc, VGpc, Q)$$

  where Q is a sequence of qualifiers specifying deviations.

- *Approximation*. The technique of approximation allows qualifiers specifying unimportant deviations to be omitted. Thus L is approximated by

  $$Lfcqa = (GGpc, VGpc, Qa)$$

  where Qa is a subsequence of Q that specifies only deviations that are judged to be important.

The remainder of this chapter describes these four techniques in detail, and shows that an arbitrary layered graph can be specified by means of them.

2.4. Factorisation

Interactions between nodes in a layered graph can be arbitrary, and can cross object slice and view boundaries. However, the organisation of the layered graph suggests the possibility of *factoring* these interactions into two orthogonal components: interactions between object slices, and interactions between views. Two object slices (views) interact if any node within the first interacts with any

node within the second; the interactions between object slices and views thus subsume all interactions between nodes. This can be illustrated by adding a new view, called the *object graph*, that specifies the interactions between object slices, and a new object slice, called the *view graph*, that specifies the interactions between views. The object and view graphs are important, because they form the basis of an effective means of specifying the interactions within a layered graph.

> **Example 2-5:** The object and view graphs of the shared database example are illustrated within the framework of the layered graph in Fig. 2-7; the top-most layer is the object graph, and the right-most vertical plane is the view graph.

The factorisation process is now described more rigorously. Let $L = (G, V, R)$ be a layered graph, where $R = \{ R1, R2, \ldots, Rk \}$ is a set of relations. Then for each x, $1 \leq x \leq k$, relations Rxg on object slices and Rxv on views can be derived from L and Rx as follows:

$$Rxg = \{ (g_1, g_2) \in G \times G \mid \exists\, v_1, v_2 \in V \text{ such that } (g_1/v_1, g_2/v_2) \in Rx \}$$

$$Rxv = \{ (v_1, v_2) \in V \times V \mid \exists\, g_1, g_2 \in G \text{ such that } (g_1/v_1, g_2/v_2) \in Rx \}$$

The Rxg are called the *object relations* and describe the object graph; the Rxv are called the *view relations* and describe the view graph.

A new relation on nodes can be derived from Rxg and Rxv by attributing to every node all the interactions in which its object slice and view are involved. The new relation is referred to as the *derived relation after factorisation*, Rxf, and is defined as follows:

$$Rxf = \{ (g_1/v_1, g_2/v_2) \in N \times N \mid (g_1, g_2) \in Rxg \text{ and } (v_1, v_2) \in Rxv \}$$

A consequence of this definition is that $Rxf \supseteq Rx$. Further define

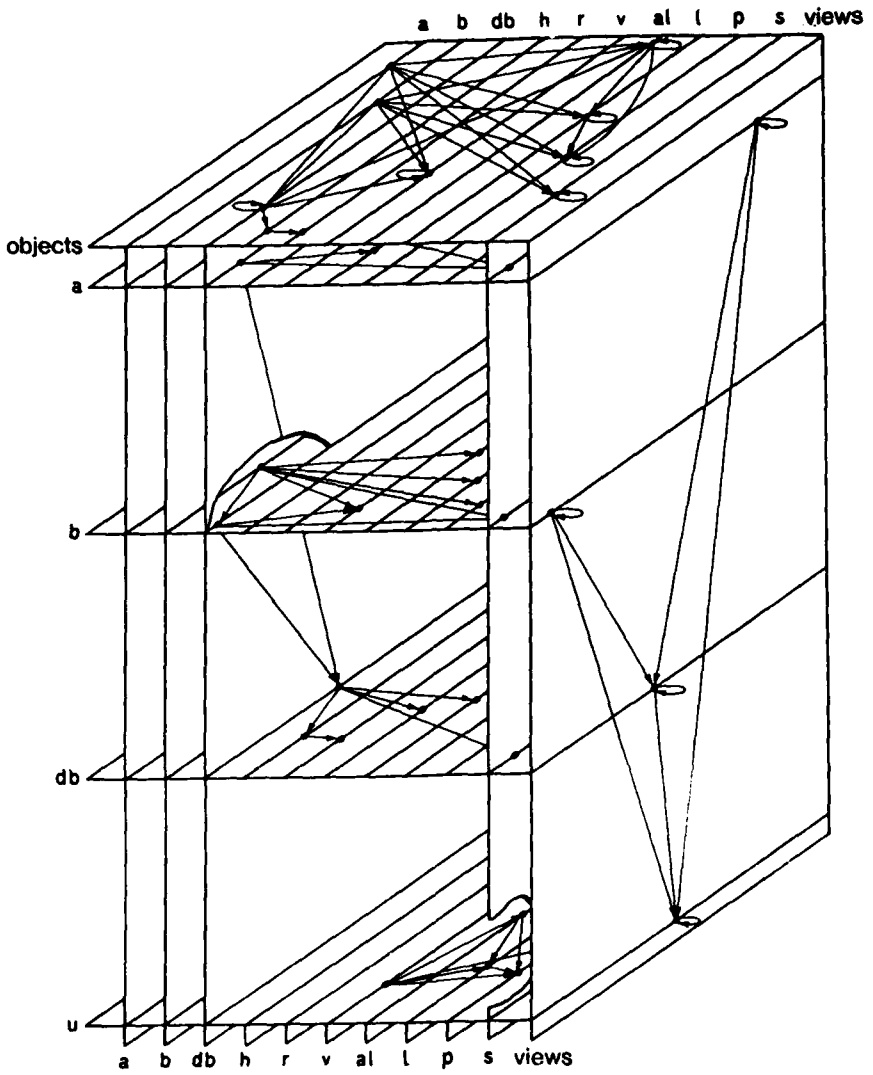


Figure 2-7: Object and View Graphs

$$Dxf = Rxf - Rx$$

Then Dxf is the set of interactions specified by Rxf that are not present in Rx, and is referred to as the set of *deviations* associated with the factorisation process applied to Rx.

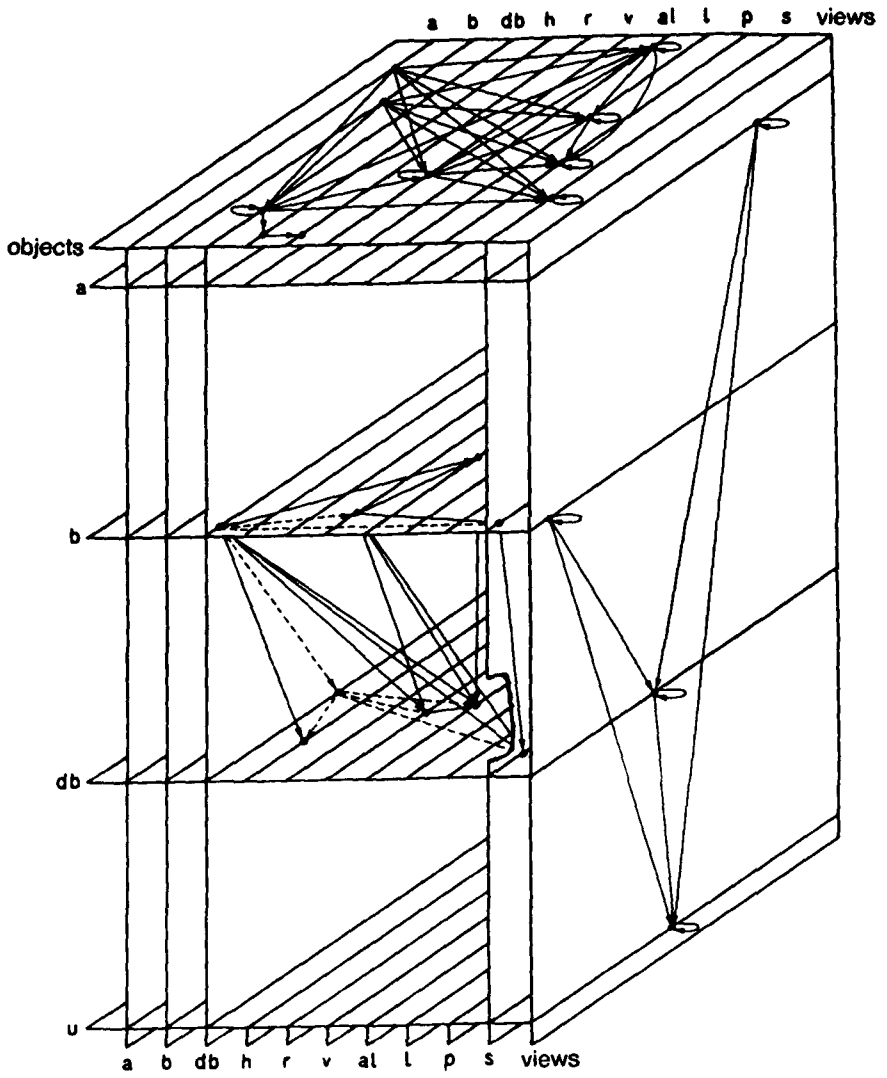


Figure 2-8: Derived Relation after Factorisation

Example 2-6: The derived relation after factorisation in the shared database example is illustrated within the framework of the layered graph in Fig. 2-8. To keep the diagram intelligible, only a few nodes are included, but all interactions between those nodes are shown. Dotted arrows represent interactions that are present in the original layered graph, as can be seen by comparison with Fig. 2-7. Solid arrows represent deviations. The deviations shown fall into three categories:

1. Interactions that cross both object slice and view boundaries.
2. Interactions between utilities.
3. The interaction between *db/b* and *alists/b*.

This is a representative sample of the types of deviations that are frequently associated with the factorisation process.

It follows from the definitions of Rxf and Dxf that

$$Rx = Rxf - Dxf$$

Since Rxf can be derived from Rxg and Rxv in a simple and direct manner, it follows that Rx can be accurately and completely specified by the triple (Rxg, Rxv, Dxf), which is referred to as the *factored form* of Rx. This is the key result concerning factorisation. It allows the layered graph

$$L = (G, V, R) = (G, V, \{ Rx \mid x = 1, 2, \ldots, k \})$$

to be specified in *factored form* as

$$Lf = (G, V, \{ (Rxg, Rxv, Dxf) \mid x = 1, 2, \ldots, k \})$$

Defining

$$Rg = \{ Rxg \mid x = 1, 2, \ldots, k \}$$
$$Rv = \{ Rxv \mid x = 1, 2, \ldots, k \}$$
$$Df = \{ Dxf \mid x = 1, 2, \ldots, k \}$$

and rearranging, yields

$$Lf = ((G, Rg), (V, Rv), Df)$$

According to definition 2-1, $GG = (G, Rg)$ and $VG = (V, Rv)$ are graphs; they are the *object graph* and the *view graph* introduced informally above. Thus the process of factorisation allows an arbitrary layered graph $L = (G, V, R)$ to be specified in factored form as

$$Lf = (GG, VG, Df)$$

The *pure* factored form mentioned in Section 2.3 consists of Lf with Df omitted:

$$Lpf = (GG, VG)$$

If $Df = \emptyset$, the layered graph can be specified accurately in pure factored form, and is termed *regular*.

Specifying a layered graph in factored form has the following important advantages:

- *Simplicity.* The object and view graphs are usually considerably smaller and simpler than the original layered graph. Provided the deviations can be characterised concisely, the factored form of the graph is therefore simpler than the original. Characterisation of deviations is discussed in Section 2.6.
- *Grouping.* Identifying and exploiting logical groupings of nodes is one of the most important factors in documenting structure. The factored form specifies interactions in terms of the two important groupings, objects and views.

The first step in specifying the structure of a layered graph is therefore to specify it in factored form, as an object graph, a view graph, and a set of deviations. The second step is to specify the object and view graphs in a clear and concise manner, as described in the next section.

2.5. Clustering

The object and view graphs produced by the process of factorisation are ordinary graphs. Direct representations of graphs, such as visibility control lists, do not work well because they are too complex in the case of large graphs, and they fail to specify grouping. The process of *clustering* allows graphs to be specified in a concise and readable fashion, largely as a result of making grouping explicit. It is similar to the process of grouping pieces of program into

modules, and results in an hierarchy much like the *system tree* in MIL 75 [DeRemer-Kron 76].

Consider a complex graph consisting of many nodes representing objects or views, and involving many interactions between the nodes. The process of clustering involves grouping the nodes into *clusters*, and replacing all interactions between nodes in different clusters by interactions between the clusters themselves. The result is a graph of clusters, each of which is a graph of nodes.

Example 2-7: Consider applying clustering to the object graph of the shared database example (Fig. 2-7). There are various ways of grouping the nodes (objects) into clusters. Since *a* and *b* in this example are users, and the other objects exist solely to provide them with the services they require, one grouping that immediately comes to mind consists of just two clusters, *users* and *servers*, as follows:

$$users = \{ a, b \}$$
$$servers = \{ db, h, r, v, al, l, p, s \}$$

Fig. 2-9 shows these clusters.

The result of replacing all interactions between nodes in different clusters with interactions between the clusters themselves is shown in Fig. 2-10.

The clustering process can be repeated recursively. If the graph of clusters is complex, it can be clustered further: this is the bottom-up approach. Alternatively, any subgraph that is complex can be clustered: this is the top-down approach. Whichever approach is used, the result is an hierarchy of graphs in which all the graphs are simple and all interactions are between siblings in the hierarchy.

Example 2-8: The cluster *servers* in Fig. 2-10 is sufficiently complex to warrant recursive clustering. Once again, many different groupings can be used; one possible result is shown in Fig. 2-11. All

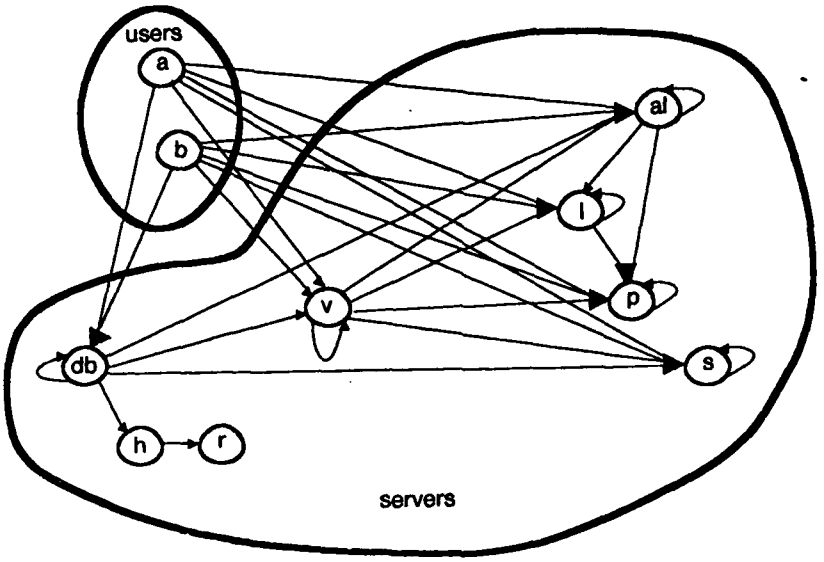


Figure 2-9: Object Graph Showing Clusters

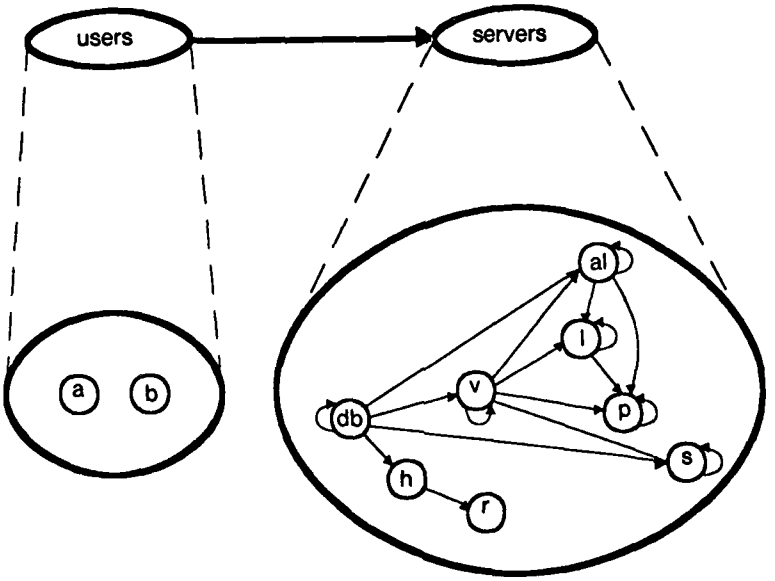


Figure 2-10: Partially Clustered Object Graph

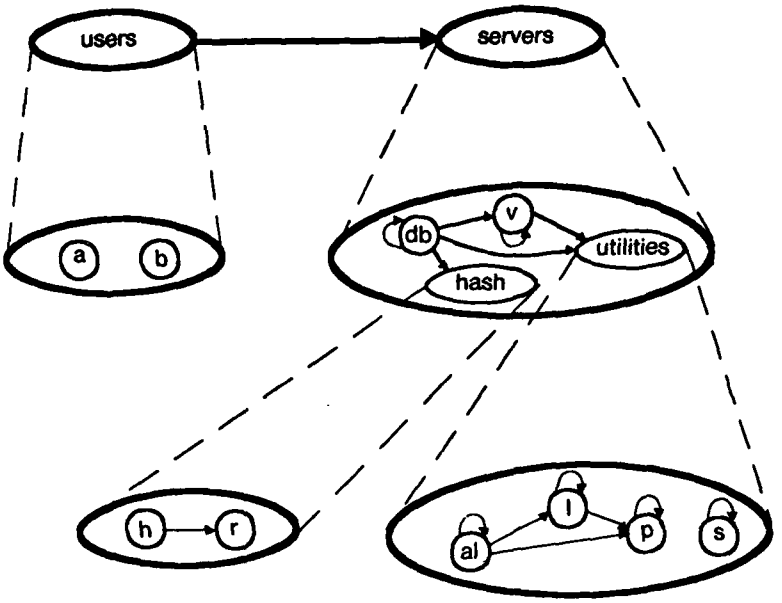


Figure 2-11: Fully Clustered Object Graph

subgraphs are now sufficiently simple that no further clustering is necessary.

The clustering process is now described more rigorously. Let $L = (G, V, R)$ be a layered graph specified in factored form as $Lf = (GG, VG, Df)$, as described in Section 2.4. The process of clustering is applied to both the object graph, $GG = (G, Rg)$, and the view graph, $VG = (V, Rv)$; let $P = (N, R)$ stand for whichever of these is currently under consideration, where $R = \{ R1, R2, \ldots, Rk \}$ is a set of relations.

The first step of the clustering process consists of choosing a partition C of N into *clusters* $C_1, C_2, \ldots, C_l$ of nodes. The nature of this partition greatly affects the process, as will become clear below. Choosing it is a difficult problem requiring an intimate understanding of the graph and the program whose structure the graph represents, and is analogous to the problem of finding an appropriate modularisation for a large program. Neither the technique of

clustering nor the grid mechanism solves this problem, but they do provide a means of using a well-chosen partition to simplify structure specification.

Once the clusters have been chosen, each relation $Rx \in R$ can likewise be partitioned into sets Rx_{ij}, $1 \leq i, j \leq l$, as follows:

$$Rx_{ij} = \{ (n_i, n_j) \in Rx \quad | \quad n_i \in C_i \text{ and } n_j \in C_j \}$$

For each i, $1 \leq i \leq l$, the pair

$$S_i = (C_i, \{ Rx_{ii} \quad | \quad x = 1, 2, \ldots, k \})$$

is a closed subgraph of P. Let

$$S' = \{ S_i \quad | \quad i = 1, 2, \ldots, l \}$$

Then for each $1 \leq x \leq k$, a relation on S', called the *subgraph relation*, Rxs, can be derived from the Rx_{ij}, $i \neq j$, to describe the relationships between these subgraphs, as follows:

$$Rxs = \{ (S_i, S_j) \quad | \quad i \neq j \text{ and } Rx_{ij} \neq \emptyset \}$$

This amounts to condensing all pairs describing relationships between the members of two clusters into a single pair relating the two clusters themselves. Note that Rxs is, by definition, non-reflexive.

Rxs is a relation on subgraphs, whereas Rx_{ii} and Rx are relations on the set of original nodes, N. A new relation on N, called the *derived relation after clustering*, Rxc, can be derived from S' and Rxs as follows:

$$Rxc = \supset Rx_{ii} \quad \cup \quad \{ (n_i, n_j) \quad | \quad 1 \leq i, j \leq l, i \neq j, \; n_i \in C_i, n_j \in C_j \text{ and } (S_i, S_j) \in Rxs \}$$

Thus, if two subgraphs are related by *Rxs*, then all nodes in the first are related to all nodes in the second by *Rxc*. A consequence of this definition is that $Rxc \supseteq Rx$. Further define

$$Dxc = Rxc - Rx$$

Then *Dxc* is the set of interactions specified by *Rxc* that are not present in *Rx*, and is referred to as the set of ***deviations*** associated with the clustering process.

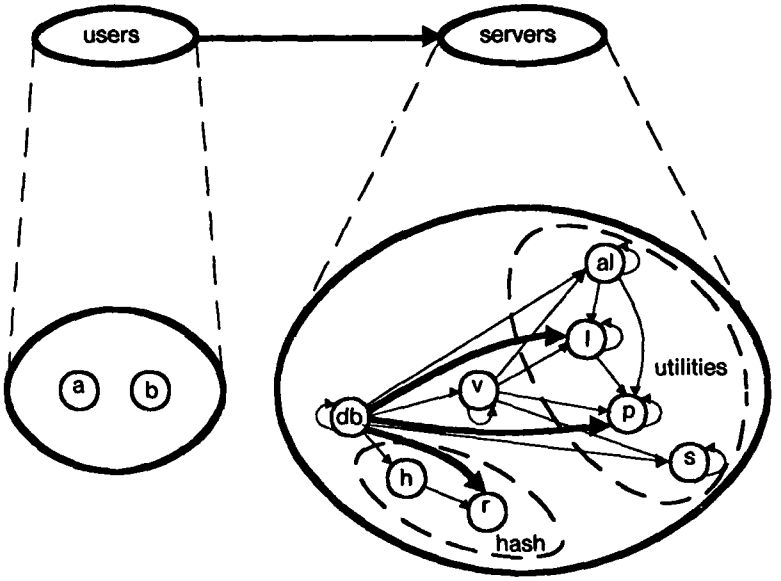


Figure 2-12: Derived Relation after Clustering

Example 2-9: Consider the clustering process described in Example 2-8 that transformed the partially clustered object graph of Fig. 2-10 into the fully clustered object graph of Fig. 2-11. The derived relation after clustering is shown within the framework of the partially clustered graph in Fig. 2-12. Light arrows represent interactions that are present in the original graph, as can be seen by comparison with Fig. 2-10. Heavy arrows represent deviations. The deviations arise because the derived relation after clustering specifies that *db* interacts with *all* nodes in clusters *hash* and *utilities*, whereas, in the original graph, it interacts with only some of them. This type of deviation is typical of the clustering process.

It follows from the definitions of Rxc and Dxc that

$$Rx = Rxc - Dxc$$

Since Rxc can be derived from S' and Rxs in a simple and direct manner, it follows that Rx can be accurately and completely specified by the triple (S', Rxs, Dxc), which is called the *clustered form* of Rx. This is the key result concerning clustering. It allows the graph

$$P = (N, R) = (N, \{ Rx \mid x = 1, 2, \ldots, k \})$$

to be specified in *partially clustered form* as

$$Pc' = (S', \{ (Rxs, Dxc) \mid x = 1, 2, \ldots, k \})$$

Defining

$$Rs' = \{ Rxs \mid x = 1, 2, \ldots, k \}$$
$$Dc' = \{ Dxc \mid x = 1, 2, \ldots, k \}$$

and rearranging, yields

$$Pc' = (S', Rs', Dc')$$

The process of clustering can be repeated recursively on any of the subgraphs within S' that are sufficiently complex to warrant it. If Pc' itself is sufficiently complex, the process can be applied to the graph (S', Rs'). For each relation $Rx \in R$, all sets of deviations arising from recursive clustering applied to Rx are united with $Dxc \in Dc'$ to form a combined set of deviations; the collection of these combined sets is denoted by Dc. When all subgraphs are sufficiently small and simple, the result,

$$Pc = (S, Rs, Dc)$$

is termed the *clustered form* of the original graph P. The *pure* clustered form consists of Pc with all deviations omitted:

$$Ppc = (S, Rs)$$

If $Dc = \emptyset$, the original graph can be specified accurately in pure clustered form, and the clustering is termed *uniform*.

Specifying graphs in fully clustered form has the following advantages:

- *Simplicity.* The hierarchy of simple graphs making up the fully clustered form is usually simpler and easier to understand than the original graph, provided the clusters are well chosen, and the deviations can be characterised concisely. Characterisation of deviations is discussed in Section 2.6.
- *Grouping.* The clusters are explicit, and the information they convey about grouping is an important aid to understanding. In fact, the hierarchical organisation of the fully clustered form is a good guide to the structure of the program, even if interactions are ignored.

Returning to the layered graph $L = (G, V, R)$ with factored form $Lf = (GG, VG, Df)$, suppose the process of clustering is applied to the object graph GG and the view graph VG until both are in fully clustered form,

$$GGc = (S_G, Rs_G, Dc_G)$$

$$VGc = (S_V, Rs_V, Dc_V)$$

Then the *clustered form* of L is

$$Lfc = ((S_G, Rs_G), (S_V, Rs_V), \{Dxf \cup Dxc_G \cup Dxc_V \mid x = 1, 2, \ldots, k\})$$

$$= (GGpc, VGpc, D)$$

The *pure* clustered form mentioned in Section 2.3 consists of Lfc with all deviations omitted:

$Lpfc = (GGpc, VGpc)$

The second step in specifying the structure of a layered graph is therefore to specify both the object graph and the view graph in fully clustered form. The next step is to characterise the set of deviations, if any, arising from factorisation and clustering. This process is described in Section 2.6.

2.6. Deviation

The processes of factorisation and clustering allow a regular, uniformly clustered layered graph to be specified accurately, clearly and concisely in pure clustered form. Most layered graphs are not regular, however, and uniform clusterings are often uninteresting.[23] The technique of deviation allows an arbitrary layered graph to be specified in pure clustered form, together with details of how the actual graph deviates from the pure clustered form. One important aspect of the technique of deviation has already been described formally: sets of deviations were derived as part of the processes of factorisation and clustering, described in Sections 2.4 and 2.5. This section discusses how deviation sets can be characterised in a clear and concise manner.

Let $L = (G, V, R)$ be an arbitrary layered graph, where $R = \{ R1, R2, \ldots, Rk \}$ is a set of relations. Suppose the pure clustered form of L is

$Lpfc = (GGpc, VGpc)$

Then a set of relations, $Rpfc = \{ R1pfc, R2pfc, \ldots Rkpfc \}$ can be derived from $Lpfc$ as described in Sections 2.4 and 2.5. The clustered form of L is then

$Lfc = (GGpc, VGpc, D)$

---

[23] The only uniform clusterings of a set of nodes that are guaranteed to exist are the two trivial clusterings: all nodes in one cluster, or each node in its own cluster.

where $D = \{ D1, D2, \ldots, Dk \}$ is a collection of *deviation sets* such that

$$Dx = Rxpfc - Rx$$

Each set of deviations, Dx, is characterised by a separate sequence of *qualifiers*, Qx. The qualifiers are intended to specify large sets of deviations in a concise and intuitive manner, and are especially designed to handle important special cases easily and gracefully. A sample of qualifiers is presented in this section; all qualifiers currently defined for the grid mechanism are presented in Section 3.3.3. The identification and definition of additional qualifiers is one of the most interesting areas of further research based on the grid mechanism, and is discussed in Chapter 6.

Most qualifiers involve sets of layered graph nodes, called *node sets*. A node set can be specified in any of the following ways:

- A node name, denoting a singleton set.
- An object slice, view or cluster name, denoting all nodes in that object slice, view or cluster.
- A set of any of the above.
- An asterisk, denoting all nodes in the graph.

Throughout the ensuing description, S_i, for some i, denotes a node set, and N denotes the set of all nodes in the layered graph.

The following is a small sample of possible qualifiers:

- The qualifier *except*(S_1, S_2) specifies deviations directly. It states that all pairs in the set $S_1 \times S_2$ are deviations.[24]
- The qualifier *only*(S_1, S_2, S_3, S_4) is the complementary form of *except*, for use when it is more convenient to specify explicitly the interactions that do occur rather than interactions that do not. It states that all pairs in the set $(S_1 \times S_2) - (S_3 \times S_4)$ are deviations.

---

[24] The negative sense of this qualifier, and some of the others, may seem confusing in this context: the qualifier *except* specifies pairs that *are* deviations. The reason for this choice is explained below.

- The qualifier *same*(S_1, S_2) is used to express succinctly that no interactions between nodes in sets S_1 and S_2 cross both object slice and view boundaries. More precisely, it states that all pairs in the set

$$\{ (g_1/v_1, g_2/v_2) \in S_1 \times S_2 \mid g_1 \neq g_2 \text{ and } v_1 \neq v_2 \}$$

  are deviations.

- The qualifier *home-view*(g, v) specifies that the unit g/v is distinguished, in that all other units in object slice g interact only with g/v; no restrictions are placed on g/v itself, however. This handles the common situation in which a single, detailed unit defines an object, all other views of that object are implemented in terms of the defining unit, and the defining unit itself is implemented in terms of other objects. More precisely, the qualifier states that all pairs in the set

$$\{ (g/v_1, g_2/v_2) \in N \times N \mid v_1 \neq v \text{ and } g_2/v_2 \neq g/v \}$$

  are deviations. As an abbreviation, g can be replaced by a set of object slices.

- The qualifier *hidden*(h), where h is an object slice, view or cluster, specifies that h is local to its innermost cluster, and is hidden from all nodes in other clusters. More precisely, it states that all pairs in the set

$$\{ (n_1, n_2) \in N \times N \mid n_2 \in h, c \text{ is the innermost cluster of } h, \text{ and } n_1 \notin c \}$$

  are deviations. A node is considered to be a member of a cluster, c, if and only if it is a member of some object slice, view or cluster that is a member of c. As a useful abbreviation, h can be replaced by a set of object slices, views or clusters.

- The qualifier *exports*(c, E), where c is a cluster and E is a subset of the members of c, is the complementary form of the *hidden* qualifier, for use in cases where it is more convenient to specify the nodes exported by a cluster rather than those hidden within it. It is equivalent to *hidden*($c - E$).

It is worth noting that, from a theoretical point of view, a sequence of *except* qualifiers suffices for specifying arbitrary deviations. The other qualifiers are introduced for the sake of clarity and convenience in practice.

Example 2-10: Each of the three categories of deviations identified in Example 2-6 can be characterised by means of a single qualifier, as follows:

1. *same*(*,*)
2. *home-view*({*v, utilities*}, *u*), where *utilities* is a cluster consisting of object slices *al, l, p* and *s* (see Fig. 2-11).
3. *except*(*db/b, alists/b*)

All the deviations shown in Fig. 2-8 can therefore be specified by the sequence

same(*,*); *home-view*({*v, utilities*}, *u*); *except*(*db/b, alists/b*)

In this case the ordering is immaterial, though in general it is important, as described below.

Example 2-11: The deviations identified in Example 2-9 can be characterised by the following sequence of qualifiers:

only(*db, utilities, db,* {*al, s*}); *hidden*(*random*)

There are also many other legitimate sequences.

Qualifiers like *except* and *only* are called *specific*, because they list specific deviations explicitly. Qualifiers like *same* and *home-view* are called *general*, because they specify general rules based on program organisation. Specific qualifiers deal with individual special cases, whereas general qualifiers deal with structuring principles. As a result, general qualifiers are preferable, and should be used whenever possible: even though the *except* qualifier suffices in all cases, it should in fact be used as little as possible.

The reason for using sequences of qualifiers, rather than sets, is that qualifiers can sometimes conflict, in which case ordering is needed resolve the conflict.

Example 2-12: Suppose exactly one interaction, (n_1, n_2), crosses *both* object slice and view boundaries. The qualifier *same*(*,*) conveys the general picture correctly, but also eliminates (n_1, n_2). The complete picture can be specified by the sequence

also(n_1, n_2); *same*(*,*)

The *also* qualifier is a special one introduced for the purpose of correcting "over-shooting" by general qualifiers: *also*(S_1, S_2) specifies that all interactions in $S_1 \times S_2$ are valid (i.e. are *not* deviations), despite qualifiers occurring later in the sequence. The general rule applying to sequences of qualifiers is that when qualifiers conflict, the one occurring earliest in the sequence applies. This rule is stated precisely in Section 3.3.3.

The essential purpose of qualifiers, in the framework of the mathematical arguments given in this chapter, is to specify deviations. From a practical point of view, however, it is usually more convenient to think of them as describing modifications to be made to a derived relation, *Rxpfc*, in order to produce the corresponding original relation, *Rx*. When using this approach, it is more helpful to read a qualifier sequence in reverse, bearing in mind the relation being specified: one begins with *Rxpfc*, and modifies it as specified by each qualifier in turn. This usage explains the names of the qualifiers, and the reason they are couched in positive rather than negative terms.

Example 2-13: The following table illustrates this approach as applied to the qualifier sequence in Example 2-12:

Qualifier	**Relation specified**
–	*Rxpfc*
same(*,*)	All interactions in *Rxpfc* that lie in a horizontal or vertical plane.
also(n_1, n_2)	All interactions in *Rxpfc* that lie in a horizontal or vertical plane, and also (n_1, n_2).

The importance of sequences of qualifiers used as described above is two-fold. First, it allows general qualifiers to be used even in cases where they over- or under-specify the deviation set. Since general qualifiers convey much more

structural information than specific ones, and in a more intuitive manner, this is a great advantage. Second, when considered in reverse order they present the reader with an increasingly more detailed and more accurate specification of a relation. This can be a considerable aid to understanding.

Returning to the layered graph $L = (G, V, R)$, the clustered form

$$Lfc = (GGpc, VGpc, D)$$

can now be rendered as

$$Lfcq = (GGpc, VGpc, Q)$$

where $Q = \{ Q1, Q2, \ldots, Qk \}$ is a collection of sequences of qualifiers characterising the deviation sets in D. This is called the *qualified clustered form* of L. Specifying a layered graph in qualified clustered form has the following advantages:

- *Aids understanding.* A graph in pure clustered form is particularly easy to understand, because object slices, views and clusters relate to each other in "regular", "uniform" ways. If one understands the object graph describing interactions between objects, for example, then one understands the structure of each separate layer, since all layers are "regular". The pure clustered form thus provides a particularly appealing first approximation to program structure as an introduction to the new reader. Once a reader has gained an understanding of this approximation, he will be able to see how the actual structure of the program differs from it by examining the qualifiers. Provided the qualifiers are not too numerous and complex, this is a good way to introduce complexity while allowing the reader to retain his higher-level impression of the structure.[25]
- *Encourages regularity and uniformity.* Loosely speaking, the less regular and uniform a clustered layered graph, the more qualifiers

[25] Experience with two large examples, discussed in Chapter 5, indicates that the number of qualifiers remains reasonably small even as program size and complexity grow. Even more important, qualifiers are carefully localised in a grid specification, as described in Section 3.3.3, so that only those that are relevant are encountered in any particular context.

> will be needed to specify its structure accurately. A program designer who is required to construct a specification of the structure of his program is therefore encouraged in a practical way to strive for regularity and uniformity whenever possible, with resultant beneficial effects on the readability of the program. Nonetheless, the technique of deviation does provide a means of handling structures that are not regular and uniform when they are truly necessary.

The third step in specifying the structure of a layered graph is therefore to characterise all deviations arising from factorisation and clustering by means of qualifiers. The final step is to remove unneeded complexity by eliminating unimportant qualifiers. This process is described in Section 2.7.

2.7. Approximation

All deviations add complexity to a structure specification, and therefore reduce its readability. This is true even of deviations that can be characterised by short sequences of qualifiers, though the simpler their characterisation the less serious their effect. Also, in many cases the deviations correct trifling inaccuracies that are not material to an understanding of the program. The technique of approximation exists in recognition of the fact that not all deviations are material, and it consists solely of ignoring them by discarding the qualifiers that characterise them. This has the effect of approximating the actual structure of the program by a structure that is more regular and/or more uniform, and that differs from the actual structure only in unimportant respects. The important advantage of this technique is that it removes unneeded complexity from a structure specification.

The programmer has the choice of which deviations to consider important enough to include, and which to ignore. It is important to note, however, that ignoring deviations can never result in omitting from the specification interactions that actually occur in the program.

Example 2-14: The *only* qualifier in Example 2-11 specifies that *db* does not use certain of the utilities. This type of deviation is typical, and is a good candidate for omission: it is usually sufficient to know that *db* uses some of the utilities, without knowing the details. The the *except* qualifier in Example 2-10 is unimportant for similar reasons, and can be discarded.

Though specifying *permitted* interactions is generally more important than specifying *actual* interactions, for reasons of documentation discussed in Section 1.3.2, actual interactions are sometimes of interest. The technique of approximation can be used as a bridge between the two. The interactions actually present in a program form a (usually proper) subset of the potential interactions permitted by the program designer. Actual interactions can therefore always be specified by adding (zero or more) qualifiers to a description of permitted interactions; omitting the additional qualifiers by the process of approximation restores the original specification of permitted interactions[26]. This fact suggests an attractive approach to program development: begin by specifying permitted interactions, but when the program is complete, add the qualifiers needed to produce a specification of actual interactions (for use in determining compilation dependencies, for example). The addition of qualifiers can be completely automated, though intelligence applied to the task might result in fewer and "better" qualifiers.

Approximation can be applied repeatedly, giving rise to a sequence of increasingly less detailed structure specifications. This sequence, when considered in reverse, presents a reader with increasingly more detailed and accurate approximations of the actual structure. Especially if the qualifiers selected for omission at each stage are logically related in some way, and their relationship is documented, such a sequence of approximations can be a useful aid to

---

[26] It is, of course, possible to arrive at specifications of permitted and actual interactions independently, in which case they may differ by more than just a few qualifiers; for example, completely different clusterings might be used. This approach seems to have limited utility.

understanding structure. The simple sequence described here could even be extended to a full abstraction hierarchy; how useful this would be is a matter for further investigation.

Chapter 3
The Grid Mechanism

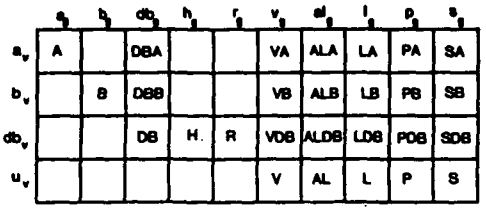
The grid is a program structuring mechanism based on the results of the previous chapter: it specifies layered graph structures using the techniques of factorisation, clustering, deviation and approximation. Section 3.1 gives a brief overview of the grid mechanism, and Sections 3.2 through 3.4 then define it precisely.

3.1. Overview

A grid corresponds to a layered graph that has undergone the processes of factorisation, clustering, deviation and approximation. It consists of a two-dimensional *matrix* of units and two *directories*, called the *object directory* and the *view directory*. The matrix specifies the organisation of units into object slices and views, the object directory is a representation of the fully clustered object graph, and the view directory is a similar representation of the fully clustered view graph.[27]

The matrix is a projection of the layered graph onto a plane orthogonal to both the object slices and the views. Each row of the matrix is called a *view*, and corresponds to a view of the graph: the units in a view describe different objects, but from the same point of view. Each column of the matrix is called a *object slice*, and corresponds to an object slice of the graph: the units in an object slice

[27] Though these representations are suitable for internal use by computers, the directories are illustrated graphically in this thesis. A complete programming environment based on the grid would provide a sophisticated user interface to allow users to manipulate directories graphically.



	a_g	b_g	db_g	h_g	r_g	v_g	al_g	l_g	p_g	s_g
a_v	A		DBA			VA	ALA	LA	PA	SA
b_v		B	DBB			VB	ALB	LB	PB	SB
db_v			DB	H	R	VDB	ALDB	LDB	PDB	SDB
u_v						V	AL	L	P	S

Figure 3-1: The Matrix

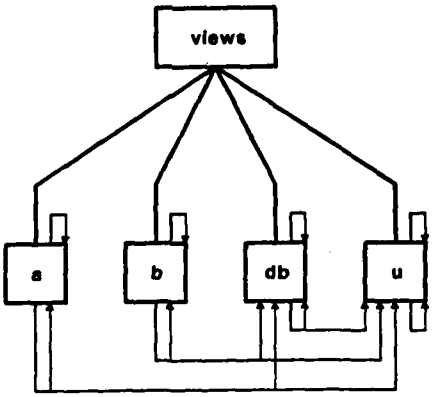


Figure 3-2: The View Directory

describe the same object, but from different points of view. The matrix thus completely captures the orthogonal partitioning of the nodes of a layered graph into object slices and views.

> **Example 3-1:** The grid matrix corresponding to the layered graph of Fig. 2-6 is shown in Fig. 3-1. The unit names correspond to those listed in examples 2-3 and 2-4. The object slice and view names are based on those listed in example 2-4, but they have been subscripted to distinguish them from each other and from node names: this is for purposes of exposition only, as meaning can always be determined from context within a grid. The matrix is obtained from the graph by dragging each node along the line of intersection between its object slice and its view until it reaches the front of the diagram. This projection loses no information, just spatial arrangement that was important for illustrative purposes but did not add in any way to the specification of structure. Note that no arrows are shown in the matrix, since the matrix specifies organisation only; all interactions are specified in the directories.

Each directory is a tree of named *nodes* representing a fully clustered graph. In the object directory, leaves represent object slices; in the view directory they represent views. In both cases, internal nodes represent clusters. Interactions and qualifiers are represented as *attributes of directory nodes*. Interactions are

between sibling nodes only,[28] and, for each kind of interaction, each node in a directory contains a list of siblings with which it interacts. Thus an arbitrary number of different kinds of interactions can be specified within a single directory. A wide variety of other information, including documentation, can also be associated with directory nodes. A directory thus resembles a MIL 75 system tree [DeRemer-Kron 76], an hierarchy of configuration files in C/Mesa [Mitchell-Maybury-Sweet 79], or an hierarchy of DF files in Schmidt's scheme [Schmidt 82], though it differs from these in some important details. The abstract syntax of the grid mechanism is described in detail in Section 3.3.

Information in a directory is highly localised: every item of information is attached to that part of the directory to which it applies. For example, sibling interactions are attached to their source nodes, and qualifiers are attached to the nodes, or even to the individual sibling interactions, to which they apply. This localisation ensures that only relevant information is encountered in any particular context, which greatly enhances readability.

> **Example 3-2:** The view directory of the shared database example is shown in Fig. 3-2, and the object directory in Fig. 3-3. Interactions are illustrated by means of arrows: arrows above their source and target nodes represent *DefUses* interactions, and arrows below their source and target nodes represent *ImpUses* interactions. Qualifiers are shown explicitly. Details of these diagrams, such as the attachment points of the qualifiers, are explained in Section 3.3.[29]

The above discussion reveals the following correspondence between the

---

[28] A leaf node is permitted to interact with itself, however.

[29] The diagrams of the directories, especially the object directory, appear rather complex, largely because they are intended to illustrate the abstract syntax of the grid in detail, not to provide an overall picture of structure. A sophisticated programming environment based on the grid could provide clearer pictures of the directories in various ways. For example, it could display selected portions of the information only, as directed by the user, or it could use perspective as in Fig. 2-11. It could also use colour to great advantage, if available.

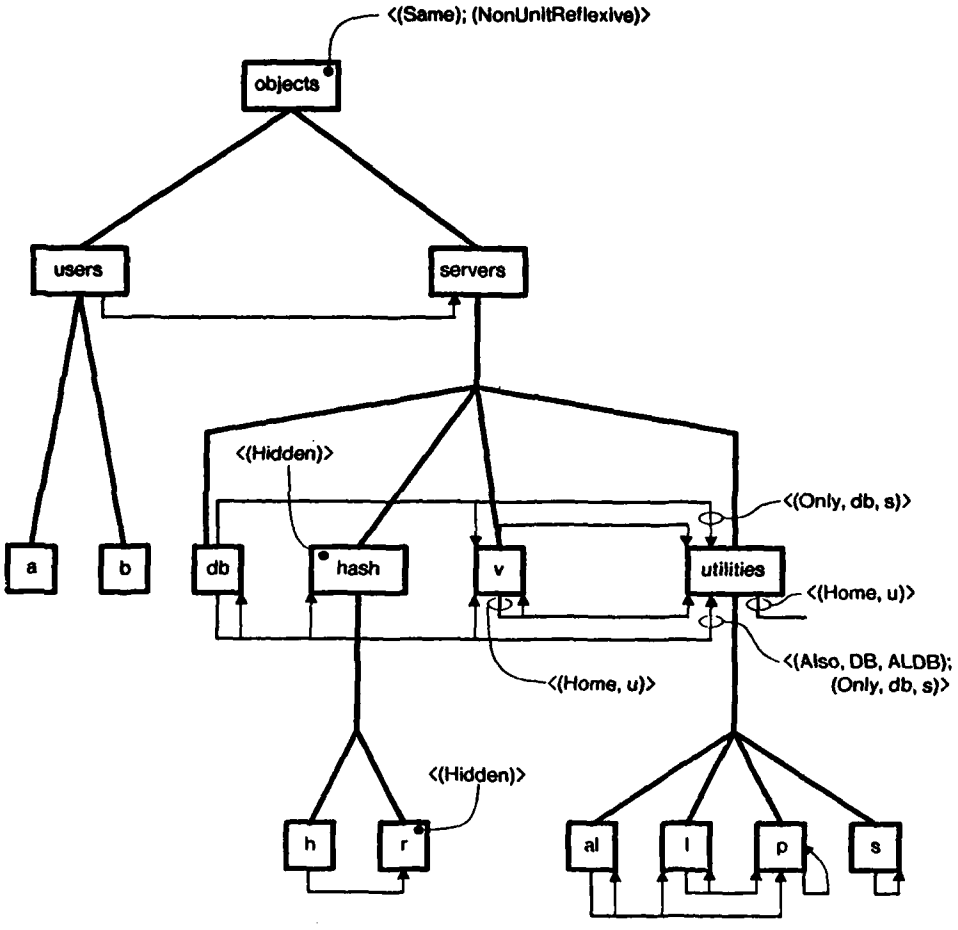


Figure 3-3: The Object Directory

abstract syntax of the grid and the techniques for specifying layered graph structures:

- The matrix derives from the dual categorisation of units according to object and view.
- The presence of two directories derives from the technique of factorisation.
- The hierarchical nature of the directories derives from the technique of clustering.
- The qualifiers derive from the technique of deviation.

There is no direct support for the technique of approximation. Approximation is achieved, instead, by the author of a structure specification omitting

unimportant details, and possibly providing multiple grid specifications of the same program that differ in the amount of detail included. The alternative approach of having all detail included in a single grid, together with a means of selecting what details to omit in a particular context, is preferable for a number of reasons. Section 6.2 discusses the issue of extending the grid mechanism to handle this approach.

From a semantic point of view, the purpose of a grid is to specify which interactions between units are valid. Consequently, the semantics of the grid can be characterised by the predicate, *valid*, that determines whether a given interaction of a given kind is specified as valid by the grid. Briefly, an interaction is valid if and only if the object directory specifies that the object slices of the two units interact in the appropriate fashion, and the view directory specifies that their views interact in the appropriate fashion. Since the object directory is a tree, any two object slices will have unique ancestors that are siblings. In the absence of qualifiers, two object slices interact if and only if an interaction of the appropriate kind is specified between these ancestors. The analogous situation is true of the view directory. Qualifiers override this "default" determination of validity in a variety of ways, and can even allow one directory to override the other. The semantics of the grid mechanism are described in detail in Section 3.4.

The grid is defined entirely in abstract terms in this thesis: an abstract syntax is defined, but no concrete syntax. This is deliberate. The grid mechanism is language independent, yet if it is to be used conveniently with a particular language, its syntax should be compatible with that of the language. Leaving the syntax undefined allows complete flexibility in selecting a suitable syntax for a particular application.

The matrix of a grid can be specified using any simple notation or naming scheme that is able to specify the position of units in the matrix. The directories

```
directory views is
    view   a        DefUses a;          ImpUses a, db, u;
    view   b        DefUses b;          ImpUses b, db, u;
    view   db       DefUses db;         ImpUses db, u;
    view   u        DefUses u;          ImpUses u;
end views;

directory objects is
    cluster users       ImpUses servers;
    cluster servers;
    Same; NonUnitReflexive;
end objects;

cluster users is
    object a;
    object b;
end users;

cluster servers is
    object db       DefUses v, utilities(Only(db, s));
                    ImpUses db, hash, v, utilities(Also(DB, ALDB), Only(db, s));
    cluster hash;
    object v        DefUses utilities;
                    ImpUses v, utilities;
    cluster utilities;
end servers;

cluster hash
    object h        ImpUses r;
    object r;
    Hidden;
end hash;

object v is
    ImpUses: HomeView u;
end v;

cluster utilities is
    object al       ImpUses al, l, p;
    object l        ImpUses l, p;
    object p        ImpUses p;
    object s        ImpUses s;
    ImpUses: HomeView u;
end utilities;

object r is
    Hidden;
end r;
```

Figure 3-4: The Directories in Textual Form

can be specified using any notation that is able to represent trees of nodes with attributes in a clear and concise manner. Qualifiers can be specified using any notation that provides simple and clear abbreviations for all important special cases.

> **Example 3-3:** The matrix of the shared database program, illustrated in Fig. 3-1, can be specified by a sequence of statements such as
>
> $$db/a = DBA;\ db/b = DBB;\ db/db = DB; \ldots$$
>
> The directories are specified in an Ada-like syntax in Fig. 3-4. All qualifiers are shown in the figure, for completeness. In practice, those discarded by the process of approximation (see example 2-14) would be omitted.

3.2. Domain of Discourse

The domain of discourse of the grid mechanism consists of three sets: *units*, *relation identifiers* and *interaction triples*. The purpose of a grid is to specify the structure of a particular set of units; i.e. how the units are organised, and how they interact. From the point of view of the grid, each unit is atomic. Throughout the descriptive passages of this thesis, each unit is assumed to be part of a program, written in some programming language; different units can be written in different languages. This interpretation is not essential to the grid, however, and does not appear in any of the formal definitions. The grid can thus be used to specify the structure of any collection of entities.

There can be many different kinds of interactions between units. For example, an object in Smalltalk [Ingalls 78] can be a *subclass* of one object, a *superclass* of another and an *instance* of a third. Each kind of interaction is characterised by a separate relation, denoted by a *relation identifier*; the italicised words in the previous sentence are examples of relation identifiers. The

grid does not make direct use of relations, but it does use the relation identifiers that denote them. For convenience, a relation identifier is sometimes said to denote a "kind of interaction", and is sometimes used as an adjective, as in "a *superclass* interaction".

An *interaction triple* describes a potential interaction of a particular kind between two units:

> **Definition 3-1:** An *interaction triple* is a triple (u_1, u_2, rid), where u_1 and u_2 are units and *rid* is a relation identifier. This triple describes an interaction of kind *rid* between units u_1 and u_2.

The following definitions introduce useful terms associated with interaction triples:

> **Definition 3-2:** Let R be a relation denoted by *rid*. Then the interaction triple (u_1, u_2, rid) is *valid for* R if and only if $(u_1, u_2) \in R$.

> **Definition 3-3:** Let $i = (u_1, u_2, rid)$ be an interaction triple. Then u_1 is called the *source* of i and u_2 is called the *target* of i.

The terms "source" and "target" are also applied to the interactions described by interaction triples.

A grid specifies the organisation and interactions of units. The set of all interactions specified by a grid is termed its set of *valid* interactions. Thus:

> **Definition 3-4:** Let *GRID* be a grid. Then an interaction triple is *valid for GRID* if and only if the interaction it describes is specified by *GRID*.

A definition of how to determine whether an arbitrary interaction triple is valid for an arbitrary grid constitutes a semantic definition of the grid mechanism. Such a definition is given in Section 3.4.

3.3. Abstract Syntax

This section describes the abstract syntax of the grid mechanism using algebraic notation. Many explanatory statements are included to indicate the meaning and purpose of the various components of the grid. Though semantic in nature, these statements should be regarded as aids to intuition, not as semantic definitions. The abstract syntax definitions are presented primarily top-down, to provide an overall picture as early as possible.

3.3.1. The Abstract Syntax of Grids

The abstract syntax of a grid is defined as follows:

Definition 3-5: A *grid* is a tuple,

$$GRID = (U, Rid, M, N, GD, VD, L, leaf)$$

where

- U is a set *units*. *GRID* specifies the organisation and interactions of these units.
- *Rid* is a set of relation identifiers. Each identifier in this set denotes a particular kind of interaction that is specified by *GRID*.
- M is a two-dimensional *matrix* of units. The rows of the matrix are called *view slices*, or simply *views*, and the columns *object slices*. The set of view slices is denoted by V, the set of object slices by G, and their union, the set of all slices, by SL. Each position in the matrix is occupied by at most one unit. The matrix thus defines two orthogonal partitions, G and V, of U.
- N is a set of *directory nodes*, often referred to merely as *nodes*.
- $GD \in N$ and $VD \in N$, $GD \neq VD$, are the *root* nodes of two *directories*, called the *object directory* and the *view directory*, respectively. The directories are disjoint trees of nodes from the set N; the tree structure is determined by the attributes of

nodes, defined in Section 3.3.2.[30] There is a one-to-one correspondence between leaf nodes in the object directory and object slices in the matrix, and between leaf nodes in the view directory and views in the matrix. The object directory specifies interactions between objects, and the view directory specifies interactions between views.

- $L \subset N$ is the set of leaf nodes in the two directories.
- *leaf*: $SL \rightarrow L$ is a bijection. It specifies the one-to-one correspondence between slices and leaf nodes, mapping each object slice in the matrix to the corresponding leaf node in the object directory, and each view in the matrix to the corresponding leaf node in the view directory.

The abstract syntax of nodes is defined in Section 3.3.2.

Example 3-4: Let *GDB* be the grid describing the structure of the shared database program introduced in Example 2-3. Unless otherwise stated, all examples in this chapter refer to *GDB*. The matrix and directories of *GDB* were illustrated in Figs. 3-1, 3-2 and 3-3; frequent reference is made to these figures throughout the ensuing examples.

According to the definition above,

$$GDB = (U, Rid, M, N, GD, VD, L, leaf)$$

where:

- U is the set of all the units shown in Fig. 3-1:

$$U = \{\ A, B, DBA, DLB, DB, H, R, VA, VB, VDB, V,\\ ALA, ALB, ALDB, AL, LA, LB, LDB, L,\\ PA, PB, PDB, P, SA, SB, SDB, S\ \}$$

- *GDB* specifies just two kinds of interactions, *DefUses* and *ImpUses*, so:

$$Rid = \{\ \text{"}DefUses\text{"}, \text{"}ImpUses\text{"}\ \}$$

- M is the matrix illustrated in Fig. 3-1. The object slices and views are:

---

[30] The term "hierarchy" is used synonymously with "tree" throughout. The properties of tree structures and the terminology commonly associated with them are used henceforth without further comment. Knuth's *Art of Computer Programming* contains a detailed and excellent exposition of such structures [Knuth 73].

$$G = \{ a_g, b_g, db_g, h_g, r_g, v_g, al_g, l_g, p_g, s_g \}$$

$$V = \{ a_v, b_v, db_v, u_v \}$$

Each object slice is actually a set consisting of all the units in its column, and each view is a set consisting of all the units in its row. For example:

$$db_g = \{ DBA, DBB, DB \}$$

$$u_v = \{ V, AL, L, P, S \}$$

- N is the set of all the directory nodes in both directories:

  $N = \{$ *objects, users, servers, users.a, users.b, servers.db, hash, v, utilities, h, r, al, l, p, s, views, views.a, views.b, views.db, u* $\}$

  Qualified names are used when necessary to identify nodes uniquely. Most subsequent examples deal with the object directory, so qualifications will be omitted from the names of object directory nodes for convenience. Thus *a*, *b*, and *db* stand for *users.a*, *users.b* and *servers.db* henceforth, whereas *views.a*, *views.b* and *views.db* are always named in full.

- GD = *objects* and VD = *views* are the roots of the directories. The partitioning of the nodes into two directories is clear from the figures.

- L is the set of all leaf nodes:

  $L = \{$ *a, b, db, h, r, v, al, l, p, s, views.a, views.b, views.db, u* $\}$

- The mapping described by the function *leaf* is clear from the correspondence between object slice, view and leaf node names. Some examples are:

  $leaf(a_g) = a$

  $leaf(a_v) = views.a$

The matrix M defines three mappings with obvious semantics:

$MU: G \times V \rightarrow U$
$MG: U \rightarrow G$
$MV: U \rightarrow V$

The mappings have the property that, for any $u \in U$,

$$MU(MG(u), MV(u)) = u$$

As a result, u can be uniquely identified by the pair g/v, where $g = MG(u)$ and $v = MV(u)$.

Example 3-5: Some examples drawn from *GDB* are:

$$MU(db_g, a_v) = DBA$$

$$MG(DBA) = db_g$$

$$MV(DBA) = a_v$$

As a result, *DBA* can be uniquely identified by the pair db_g/a_v. Note that $MU(a_g, b_v)$ is undefined in *GDB*, as there is no *b* view of *a*.

It is sometimes necessary to identify which of the directories contains a particular node. Hence the following definition:

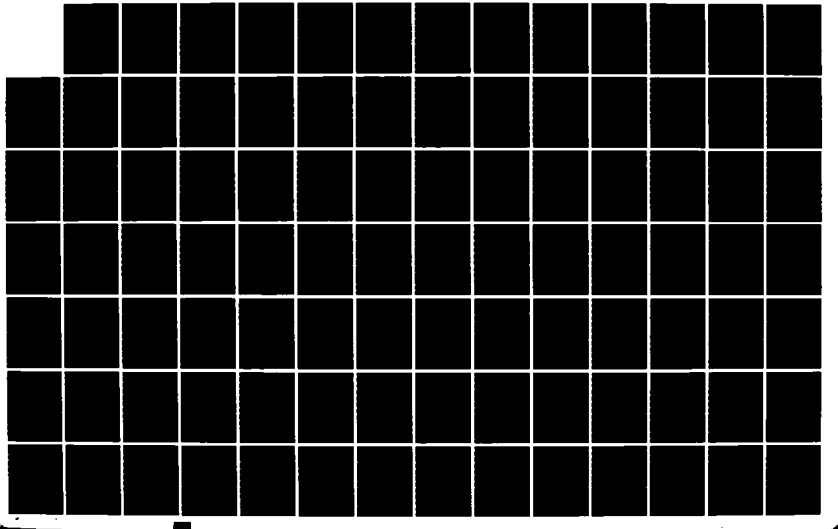
Definition 3-6: Let *n* be a directory node. Then *n* is *contained in* the object directory if and only if *n* is a descendant of *GD*, and *n* is *contained in* the view directory if and only if *n* is a descendant of *VD*.

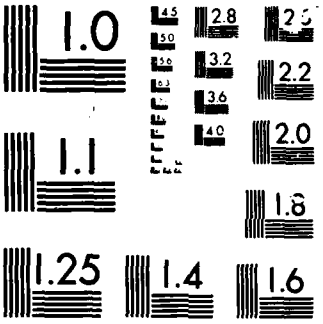
The fact the the directories are disjoint guarantees that any node is contained in just one directory.

The *leaf* function is defined on slices; for convenience, it is extended to units, as follows:

AD-A166 937 A NEW PROGRAM STRUCTURING MECHANISM BASED ON LAYERED GRAPHS(U) STANFORD UNIV CA DEPT OF COMPUTER SCIENCE H L OSSHER DEC 84 STAN-CS-85-1070 MDA903-80-C-0432 2/3

UNCLASSIFIED F/G 9/2 NL





MICROCOPY CHART

Definition 3-7: Let $u \in U$ be a unit. Then

$$leaf_{GD}(u) = leaf(MG(u))$$

and

$$leaf_{VD}(u) = leaf(MV(u))$$

are the *leaf nodes corresponding to u* in the object and view directories, respectively.

Example 3-6: Some examples drawn from *GDB* are:

$$leaf_{GD}(DBA) = db$$

$$leaf_{VD}(DBA) = views.a$$

3.3.2. The Abstract Syntax of Directories

The directories are hierarchies of nodes. The leaf nodes of the directories represent slices, which in turn contain units. It is often convenient to think of the slices and their units as "hanging" from the directories, and to be able to refer to all the slices or units that "fall under" a particular directory node. Hence the following definition:

Definition 3-8: A slice, *sl*, is said to be *subsidiary* to a directory node *n* if and only if $leaf(sl)$ is a descendant of *n*.[31] A unit, *u*, is said to be *subsidiary* to a directory node *n* if and only if $leaf_{GD}(u)$ or $leaf_{VD}(u)$ is a descendant of *n*.

A corollary of this definition and the one-to-one correspondence between leaf nodes and slices, is that all object slices and all units are subsidiary to *GD*, and all views and all units are subsidiary to *VD*.

Example 3-7: In *GDB*, object slices h_g and r_g are both subsidiary to *hash*. As a result, units *H* and *R* are subsidiary to *hash*. These

---

[31] A node is always considered to be a descendant (similarly ancestor) of itself. Thus *sl* is always subsidiary to $leaf(sl)$.

same object slices and units are also subsidiary to *servers* and to *objects*, but not, for example, to *users* or *utilities*.

The directories serve two purposes: to specify the hierarchical organisation of objects and views into clusters, and to specify interactions. The first purpose is achieved by the tree structure of the directories, and the second by attributes of nodes specifying interactions between nodes. Though interactions between nodes are different from the interactions between units described in Section 3.2, they are interactions nonetheless, and it is convenient to use similar terminology for them. Accordingly, the terms *source* and *target* are used in the obvious way, and the notion of an interaction triple is extended to interactions between nodes as follows:

Definition 3-9: A *node interaction triple* is a triple (n_1, n_2, *rid*), where n_1 and n_2 are nodes in the same directory, and *rid* is a relation identifier. This triple describes an interaction of kind *rid* between nodes n_1 and n_2.

The most usual case is that in which both nodes are leaves of a directory:

Definition 3-10: A *leaf interaction triple* is a node interaction triple (n_1, n_2, *rid*), in which both n_1 and n_2 are leaf nodes contained in the same directory.

The qualifications "node" or "leaf" are sometimes omitted when they are clear from context.

The attributes of nodes that specify interactions fall into two categories:

- *Sibling interactions* specifying interactions between sibling nodes in the directories. They correspond to subgraph relations produced by the process of clustering described in Section 2.5, and to the "sibling accessibility links" of MIL 75 [DeRemer-Kron 76].
- *Qualifiers* specifying deviations.

Sibling interactions are a special case of node interactions. Reflexive sibling interactions are allowed for leaf nodes, but not for internal nodes.

Example 3-8: The arrows in Fig. 3-3 specify sibling interactions. For example, the arrow from *users* to *servers* specifies the sibling interaction (*users, servers,* "*ImpUses*"). The arrow from *al* to itself specifies the sibling interaction (*al, al,* "*ImpUses*"), which is an example of a reflexive sibling interaction involving a leaf node; it is needed to specify that some views of *al* are implemented in terms of other views of *al*. Qualifiers are shown explicitly in the figure.

In the absence of qualifiers, the validity of a unit interaction is determined solely by the sibling interactions in the two directories. A sibling interaction (s_1, s_2, *rid*) in directory D implies that all unit interactions of the form (u_1, u_2, *rid*), where u_1 is subsidiary to s_1 and u_2 is subsidiary to s_2, are valid according to D. A specific unit interaction is valid according to the grid if and only if it is valid according to both directories. This interpretation is derived from the techniques of factorisation and clustering.

Example 3-9: If the qualifiers in Fig. 3-3 are ignored, the sibling interaction (*users, servers,* "*ImpUses*") implies the unit interactions (*A, R,* "*ImpUses*") and (*A, S,* "*ImpUses*"), among others. The reflexive sibling interaction (*al, al,* "*ImpUses*") implies the unit interaction (*ALA, AL,* "*ImpUses*"), among others. If qualifiers are ignored, the sibling interactions of kind "*ImpUses*" in both directories imply 191 unit interactions.

Qualifiers are needed when some of the unit interactions implied by the sibling interactions are in fact not valid. Such unit interactions are called *deviations*, and the purpose of qualifiers is to specify them. Some qualifiers are *local*, and specify only deviations associated with specific sibling interactions; others are *global* and specify deviations involving an entire subtree, and often an entire directory.

Example 3-10: The *Hidden* qualifiers in Fig. 3-3 specify that the unit interaction (*A, R,* "*ImpUses*") of Example 3-9 is in fact not valid. The interaction (*A, S,* "*ImpUses*") remains valid despite the

qualifiers. In all, the qualifiers specify that only 45 of the 191 unit interactions implied by the sibling interactions are in fact valid. The *Only* and *Also* qualifiers in the figure are local; all the others are global.

The considerations above motivate the following definition of directory nodes:

Definition 3-11: A *directory node*, or simply *node*, of grid

$GRID = (U, Rid, M, N, GD, VD, L, leaf)$

is a triple $n = (p, IL, GQn)$, where

- $p \in N$ is the parent of n in directory that contains it. If $n = GD$ or $n = VD$, then p is undefined. Otherwise there is a sequence of nodes $n_1 = n, n_2, \ldots, n_k$ $(k \geq 2)$, with $p(n_k)$ undefined and $p(n_i) = n_{i+1}$ $(1 \leq i < k)$. This sequence is referred to as the *path* of n. This definition of p guarantees that the directories are trees, and, since $GD \neq VD$, that they are disjoint.
- IL is a set of *interaction lists* associated with n, specifying all sibling interactions of which n is the source. There is one list in IL for each kind of interaction of which n is the source. The node n is called the *owner* of every interaction list in IL. Interaction lists owned by different nodes are assumed to be distinct, so each interaction list has a unique owner.
- GQn is a sequence of *global qualifiers* associated with n that apply to all unit interactions whose sources, or in some cases whose targets, are subsidiary to n.

Interaction lists and qualifiers are defined below.

Example 3-11: The node *db* in Fig. 3-3 is the triple

(*servers*, *IL*, ⟨⟩)

where ⟨⟩ denotes the null sequence, and *IL* consists of two interaction lists, one for "*DefUses*" and one for "*ImpUses*". The "*DefUses*" list specifies sibling interactions with *v* and *utilities*, and the "*ImpUses*" list specifies sibling interactions with *db* itself, *hash*, *v* and *utilities*. The sequence of global qualifiers, *GQn*, is null, because the node has no such qualifiers associated with it. This situation is common.

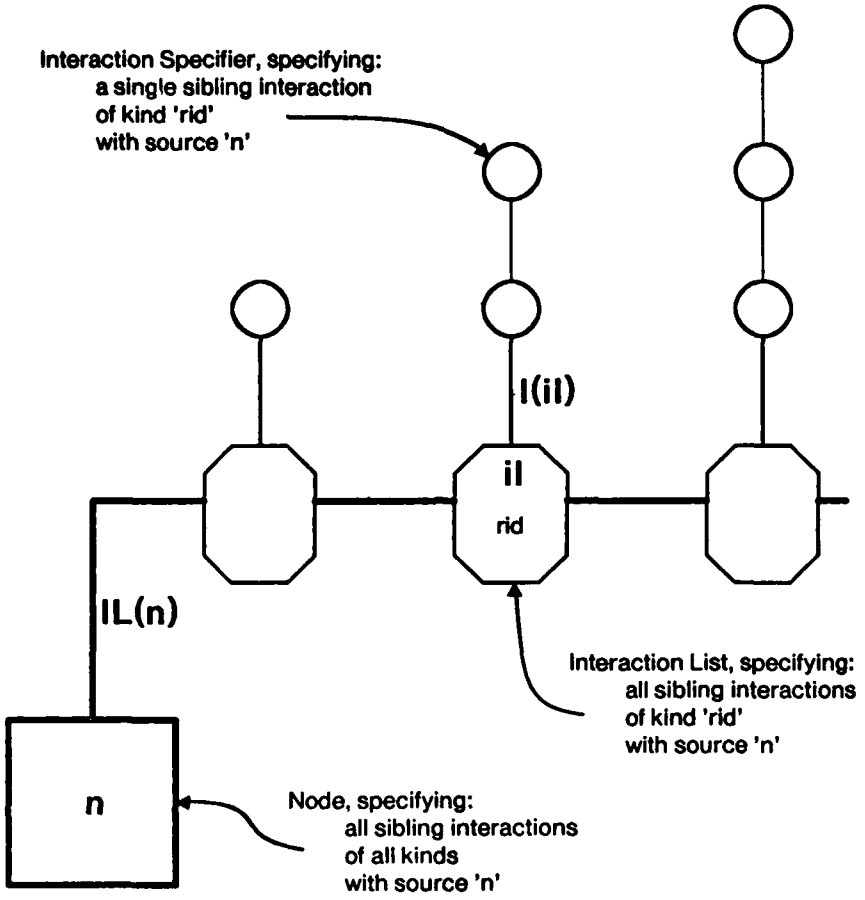
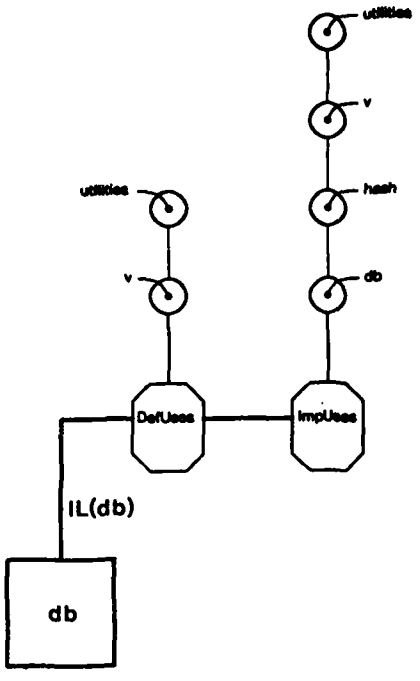


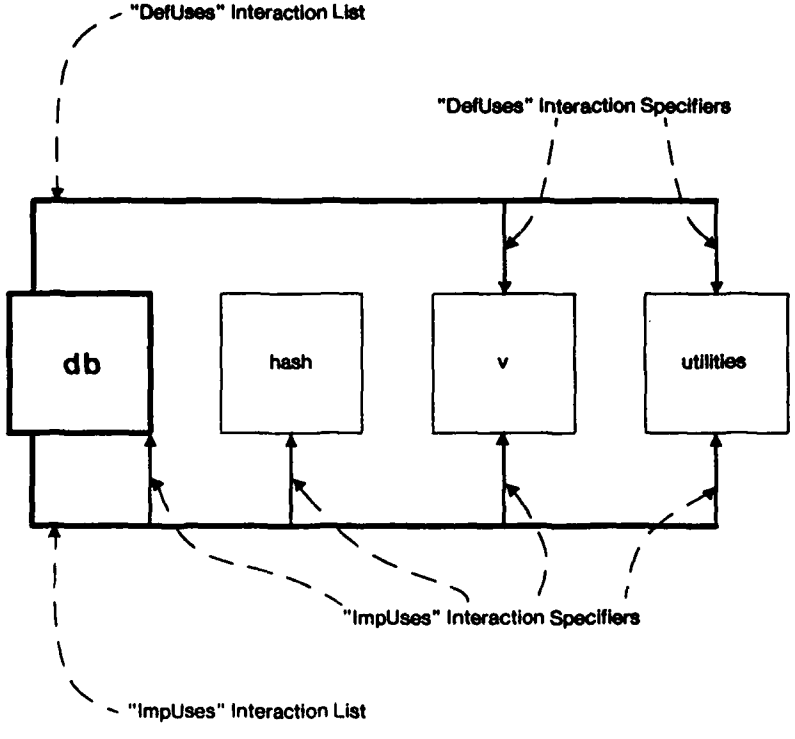
Figure 3-5: Specification of Sibling Interactions

The task of specifying sibling interactions is the responsibility of the interaction lists in *IL*. There is one interaction list for each kind of interaction. Each interaction list, in turn, consists of a number of *interaction specifiers*, each specifying a single sibling interaction. This situation is illustrated in Fig. 3-5, to place the ensuing definitions of interaction lists and interaction specifiers in perspective.

Example 3-12: The interaction lists and interaction specifiers of *db* are shown in Fig. 3-6(a). Qualifiers have been omitted temporarily. While this form of illustration makes the details clear, it is too cumbersome for use in full examples. A more compact illustration of



(a) Full Form



(b) Compact Form

Figure 3-6: Sibling Interactions of *db*

the same information is shown in Fig. 3-6(b). Each interaction list belonging to a node is represented by a separate heavy line emanating from that node, and each interaction specifier is represented by a light arrow leading from its interaction list to the appropriate target node. In all diagrams of the directories of *GDB*, interaction lists emanating from the tops of nodes specify "*DefUses*" interactions, and interaction lists emanating from the bottoms of nodes specify "*ImpUses*" interactions. These conventions were used in Figs. 3-2 and 3-3, and should be used to interpret all arrows occurring in those figures.

Interaction lists are defined as follows:

Definition 3-12: An *interaction list* with owner n is a tuple $il = (rid, I, Ql, GQl)$, where

- $rid \in Rid$ is a relation identifier indicating the kind of interactions specified by il.
- I is a set of *interaction specifiers*, specifying all sibling interactions of kind *rid* of which n is the source. The node n is said to be the owner of every interaction specifier in I, and *rid* is said to be the *kind* of every interaction specifier in I. Each interaction specifier has a unique owner and kind.
- Ql is a sequence of *local qualifiers* that apply to all the sibling interactions specified by I.
- GQl is a sequence of *global qualifiers* that apply to all *rid* interactions whose source is subsidiary to n.

Interaction specifiers and qualifiers are defined below. For any node, n, the set $II(n)$ has the property that no two interaction lists in the set have the same *rid*. There need not be an interaction list corresponding to each $rid \in Rid$, however; this merely reflects the fact that not all nodes are the sources of sibling interactions of all kinds.

Example 3-13: The "*ImpUses*" interaction list of *db* is the tuple

("*ImpUses*", I, $\langle\rangle$, $\langle\rangle$)

where I contains four interaction specifiers, one for each of the sibling interactions with source *db*:

(*db*, *db*, "*ImpUses*"), (*db*, *hash*, "*ImpUses*"),
(*db*, *v*, "*ImpUses*"), (*db*, *utilities*, "*ImpUses*")

Both sequences of qualifiers are null in this case.

Example 3-14: The "*ImpUses*" interaction list of *utilities* is the tuple

("*ImpUses*", $\emptyset$, $\langle\rangle$, $\langle$(*Home*, *u*)$\rangle$)

In this case, the set of interaction specifiers is empty, as *utilities* is the source of no sibling interactions. The interaction list exists solely to provide an attachment point for the global qualifier (*Home*, *u*) that applies to all "*ImpUses*" interactions whose sources are subsidiary to *utilities*.

An interaction specifier specifies a single sibling interaction, and any qualifiers associated with it:

Definition 3-13: An *interaction specifier* with owner n and kind *rid* is a pair $is = (s, Qi)$, where

- $s \in N$ is a sibling of n, specifying the sibling interaction (n, s, *rid*). If n is a leaf node, s can be equal to n, specifying a reflexive sibling interaction.
- Qi is a sequence of *local qualifiers* that apply only to the sibling interaction specified by s.

The abstract syntax of qualifiers is defined in Section 3.3.3.

Example 3-15: The interaction specifier with owner *db* and kind "*ImpUses*" that specifies the sibling interaction (*db*, *hash*, "*ImpUses*") is

(*hash*, $\langle\rangle$)

There are no local qualifiers in this case. The interaction specifier with owner *db* and kind "*ImpUses*" that specifies the sibling interaction (*db*, *utilities*, "*ImpUses*") is

(*utilities*, ((*Also*, *DB*, *ALDB*); (*Only*, *db*, *s*)))

3.3.3. The Abstract Syntax of Qualifiers

The reason for the complex, three-tier scheme for specifying sibling interactions, involving nodes, interaction lists and interaction specifiers, is that it provides four different and useful attachment points for qualifiers. These are illustrated in Fig. 3-7. They are used to determine to which interaction triples each qualifier applies, and in what order qualifiers are examined. Details are given in Section 3.4.

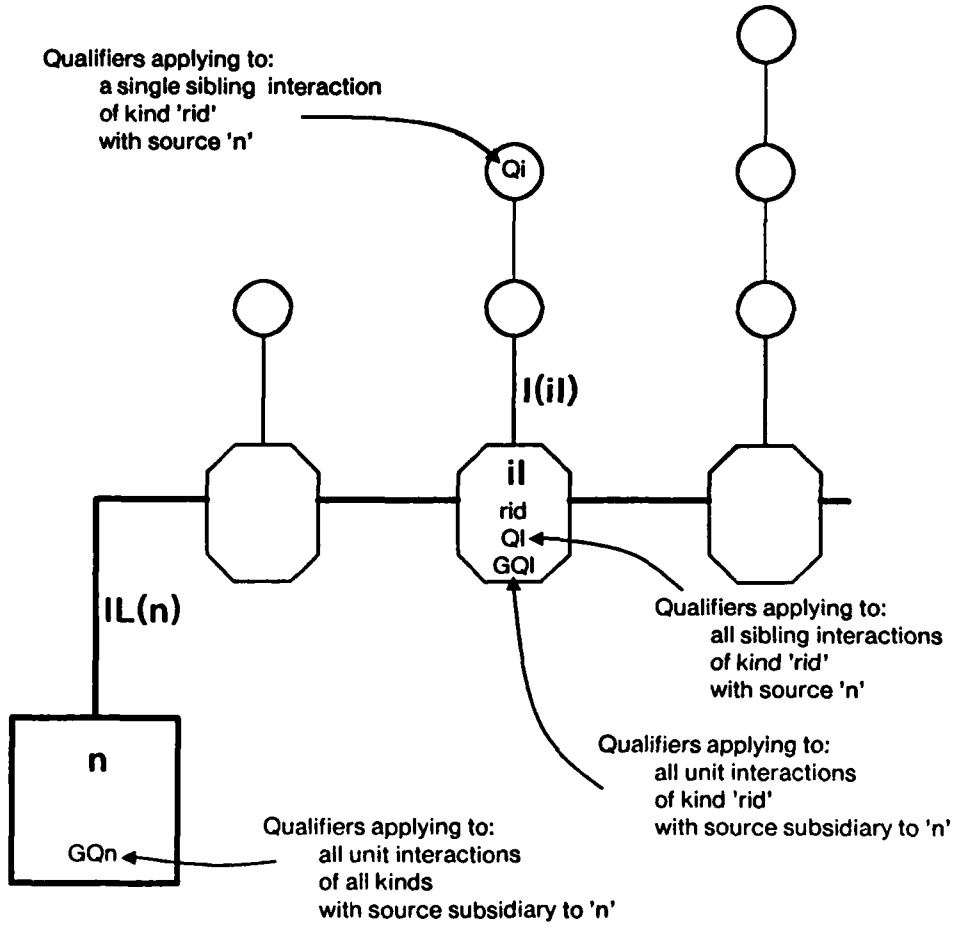
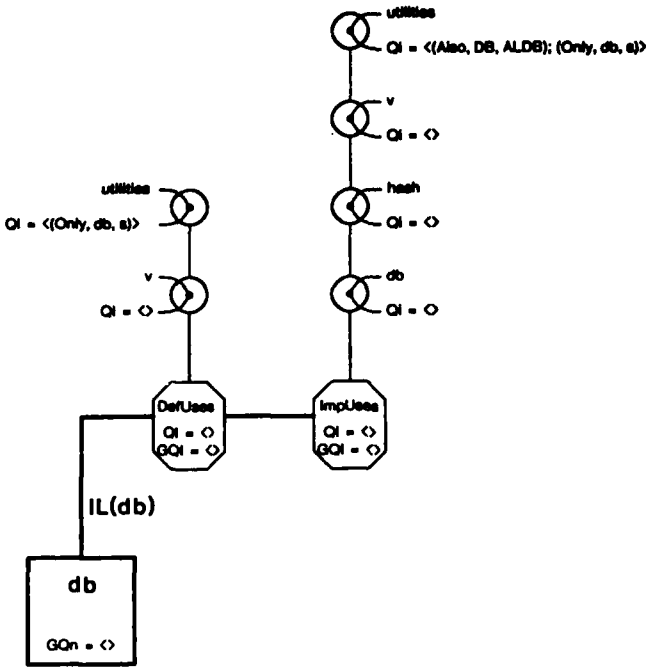
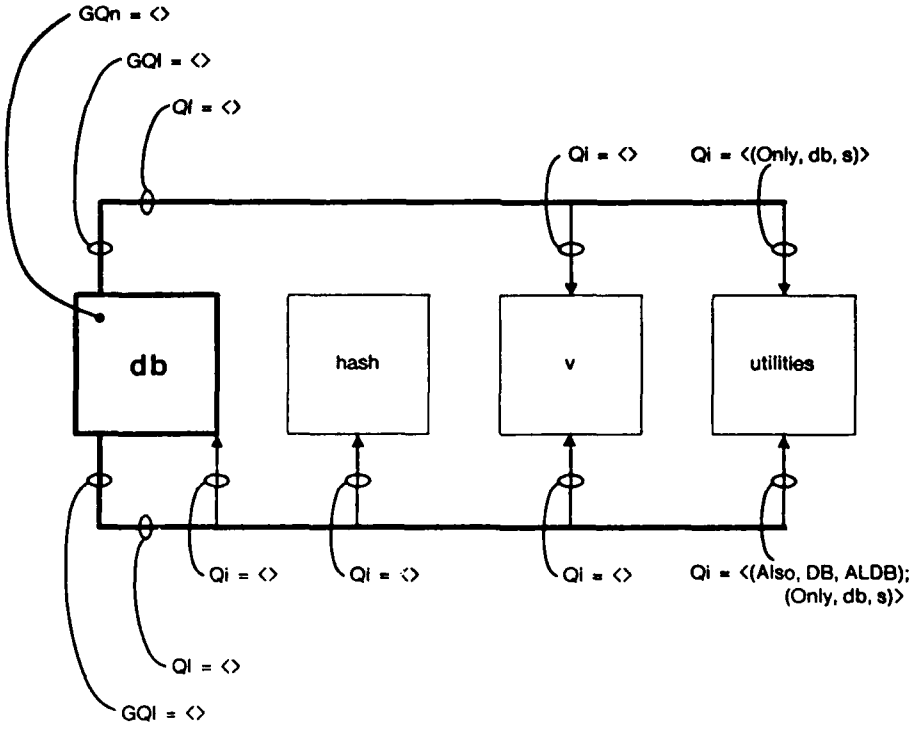


Figure 3-7: Attachment Points for Qualifiers



(a) Full Form



(b) Compact Form

Figure 3-8: Qualifiers of *db*

Example 3-16: Fig. 3-8(a) shows the interaction lists associated with node *db*, with attachment points for qualifiers indicated. Only one sequence of qualifiers is actually present however. The same information is shown in compact form in Fig. 3-8(b). This form was used in Fig. 3-3 of the full object directory, which includes further examples of attachment points: the *Same*, *NonUnitReflexive* and *Hidden* qualifiers are attached to nodes, the *Home* qualifiers are attached to interaction lists, and the *Also* and *Only* qualifiers are attached to an interaction specifier.

Most qualifiers involve ***unit sets***, and it is convenient to define these before defining qualifiers themselves. A unit set specifies a set of units, but the specification can be in terms of directory nodes; unit sets are thus directly analogous to the *node sets* of Section 2.6.

Definition 3-14: A *unit set* is a member of $2^{(U \cup N)}$. In other words, it is a set of units and/or directory nodes.

Unit set membership is defined as follows:

Definition 3-15: Let $S = \{ s_1, s_2, \ldots, s_k \}$ be a unit set, and let $u \in U$ be a unit. Then $u \in S$ if and only if $\exists\ 1 \leq i \leq k$ such that $u = s_i$ (if s_i is a unit) or u is subsidiary to s_i (if s_i is a node).

The asterisk used in node sets in Section 2.6 to denote all units is replaced here by *GD* or *VD*, the roots of the directories. Object slices and views are denoted by the leaf nodes corresponding to them in the directories. For convenience, the braces are usually omitted from a unit set consisting of a single unit or node: thus $\{ s_1 \}$ is usually written as just s_1.

Example 3-17: In *GDB*, the unit set $\{v\}$, or just v, corresponds to object slice v_g; its members are *VA*, *VB*, *VDB* and *V*. The unit set $\{hash\}$, or just *hash*, specifies all units subsidiary to node *hash*: its members are *H* and *R*. The unit set $\{v, hash\}$ is the union of the two

unit sets above. The unit sets {*objects*} and {*views*} are equal, and consist of all the units in the program.

The qualifiers themselves are now defined. In all cases, S_1 and S_2 denote unit sets. In the comments describing the intent of the qualifiers, u_i is assumed to be a member of S_i if S_i appears in the qualifier and of U otherwise, and *rid* denotes any interaction kind to which the qualifier applies. The informal descriptions of intent are made precise in Section 3.4.

Definition 3-16: Let S_1 and S_2 denote arbitrary unit sets. Then a *qualifier* is any of the following:

- An *Except* qualifier,

  $$q = (Except, S_1, S_2)$$

  This qualifier removes interactions of the form (u_1, u_2, rid).

- An *Only* qualifier,

  $$q = (Only, S_1, S_2)$$

  This qualifier removes interactions that are *not* of the form (u_1, u_2, rid).

- An *AlsoHere* qualifier,

  $$q = (AlsoHere, S_1, S_2)$$

  This qualifier adds interactions of the form (u_1, u_2, rid).

- An *Also* qualifier,

  $$q = (Also, S_1, S_2)$$

  This qualifier adds interactions of the form (u_1, u_2, rid), and in addition overrides any specifications to the contrary that occur in the *other* directory. It is used to avoid duplication in both directories of qualifiers adding interactions between individual units.

- A *Same* qualifier,

  $$q = (Same)$$

  This qualifier removes interactions of the form (u_1, u_2, rid) in which u_1 and u_2 do not belong to either the same object slice or the same view, or both.

- A *SameOther* qualifier,

  $q = (SameOther)$

  If this qualifier occurs in the object directory, then it removes interactions (u_1, u_2, rid) in which u_1 and u_2 do not belong to the same view. Similarly with "object" and "view" exchanged.

- A *Home* qualifier,

  $q = (Home, l)$

  where l is a leaf node representing an object slice or view. If this qualifier occurs in the object directory, it is referred to as a *home-view* qualifier, and l is a leaf node in the view directory specifying a view. It removes all interactions of the form (u_1, u_2, rid) *except* those for which u_1 is in the view denoted by l, or u_1 and u_2 are in the same object slice and u_2 is in the view denoted by l. Similarly with "object" and "view" exchanged.

- A *NonUnitReflexive* qualifier,

  $q = (NonUnitReflexive)$

  This qualifier removes all interactions of the form (u_1, u_2, rid) in which $u_1 = u_2$.

- A *Hidden* qualifier,

  $q = (Hidden)$

  This qualifier specifies that the node to which it belongs is local to its parent and is hidden from all nodes that are not subsidiary to its parent.

Most of these qualifiers correspond in meaning, and largely in form, to those introduced in Chapter 2. The main difference in form is that some of the unit sets are deduced from the attachment point of the qualifier in the grid, and hence are not explicit components of the qualifier. There is no direct analogy of the *Exports* qualifier in the grid: it is realised by means of *Hidden* qualifiers.[32]

---

[32] A *concrete* syntax of the grid could contain an *Exports* qualifier, that would simply be translated into the appropriate *Hidden* qualifiers.

Example 3-18: Fig. 3-3 contains examples of several of the qualifiers above.

Many other qualifiers are possible, and exploring them is one of the most interesting areas of further research associated with the grid. This issue is discussed further in Chapter 6.

Qualifiers add considerable complication to a grid, so it is sometimes convenient to deal with qualifier-free grids:

Definition 3-17: A grid is termed *ideal* if all sequences of qualifiers within it are null.

The close correspondence between the grid mechanism and the techniques described in Chapter 2 ensures that the *pure* clustered form of a layered graph can be specified directly as an ideal grid. As a result, an ideal grid is also an accurate representation of a regular, uniformly clustered layered graph.

3.4. Semantics

This section defines the semantics of the grid mechanism precisely by defining the predicate

$$valid_{GRID}: U \times U \times Rid \rightarrow Boolean$$

that determines whether a given interaction triple is valid for *GRID*. Some useful concepts and terminology used in the definition are introduced first in Section 3.4.1. Since qualifiers cause considerable semantic complexity, the semantics of ideal grids with no qualifiers are defined next, in Section 3.4.2. Subsequent sections then deal with the semantics of grids that include qualifiers. Section 3.4.7 contains three complete examples based on *GDB*, showing the steps involved in determining the validity of three different interaction triples.

3.4.1. Concepts and Terminology

Let $GRID = (U, Rid, M, N, GD, VD, L, leaf)$ be the grid whose semantics are to be specified. For any unit $u \in U$, the matrix associates a unique object slice and a unique view with u, and the function *leaf* associates unique leaf nodes with the object slice and view. These leaf nodes are important because they characterise u within the directories.

Definition 3-18: Let $u \in U$ be a unit. Then the *leaf pair* of u is the pair of nodes

$$(leaf_{GD}(u), leaf_{VD}(u))$$

Example 3-19: Consider the grid, *GDB*, describing the structure of the shared database program. The leaf pair of *DB* is (*db*, *views.db*). The leaf pair of *PDB* is (*p*, *views.db*).

The fact that the directories are trees guarantees that any two nodes in a directory have a unique *lowest common ancestor*.

Example 3-20: In *GDB*, the lowest common ancestor of *db* and *p* is *servers*. The lowest common ancestor of *a* and *s* is *objects*.

This property of directories leads to the following definition, which plays a major role in the semantics of grids:

Definition 3-19: Let n_1 and n_2 be leaf nodes in a directory. Then the *sibling ancestor pair* of (n_1, n_2) is defined as follows:

- If $n_1 = n_2 = n$, then the sibling ancestor pair is (n, n).
- Otherwise, the sibling ancestor pair is the pair (s_1, s_2), where s_1 is an ancestor of n_1, s_2 is an ancestor of n_2, and both s_1 and s_2 are children of the lowest common ancestor of n_1 and n_2.

The tree structure of the directories ensures that the sibling ancestor pair of any pair of leaf nodes exists and is unique.

Example 3-21: The sibling ancestor pair of (*db*, *p*) is (*db*, *utilities*). The sibling ancestor pair of (*a*, *s*) is (*users*, *servers*). The sibling ancestor pair of (*views.db*, *views.db*) is (*views.db*, *views.db*).

The notion of sibling ancestor pair is extended to units, as follows:

Definition 3-20: Let D be a directory, let u_1 and u_2 be units, and let $n_1 = leaf_D(u_1)$ and $n_2 = leaf_D(u_2)$ be the leaf nodes corresponding to u_1 and u_2 in D. Then the sibling ancestor pair of (u_1, u_2) in D is the sibling ancestor pair of (n_1, n_2).

Any pair of units therefore has a unique sibling ancestor pair in each directory.

Example 3-22: The sibling ancestor pair of (*DB*, *PDB*) in the object directory is (*db*, *utilities*), and in the view directory is (*views.db*, *views.db*). The sibling ancestor pair of (*A*, *S*) in the object directory is (*users*, *servers*), and in the view directory is (*views.a*, *u*).

Suppose (u_1, u_2, rid) is an interaction triple, D is a directory, and (s_1, s_2) is the sibling ancestor pair of (u_1, u_2) in D. The significance of this pair is that, if (u_1, u_2, rid) is valid for D, there will be an interaction specifier attached to s_1 that names s_2 as the target sibling, and this interaction specifier will be in the *rid* interaction list associated with s_1. These considerations lead to the following definitions:

Definition 3-21: Let (n_1, n_2, rid) be a leaf interaction triple, and let (s_1, s_2) be the sibling ancestor pair of (n_1, n_2). Then $IL(s_1)$ is the set of interaction lists associated with s_1. If

$$\exists\ il = (r, I, Ql, GQl) \in IL(s_1) \text{ such that } r = rid$$

then *il* is called the *active interaction list* of (n_1, n_2, rid). If no such *il* exists, then the active interaction list of (n_1, n_2, rid) does not exist.

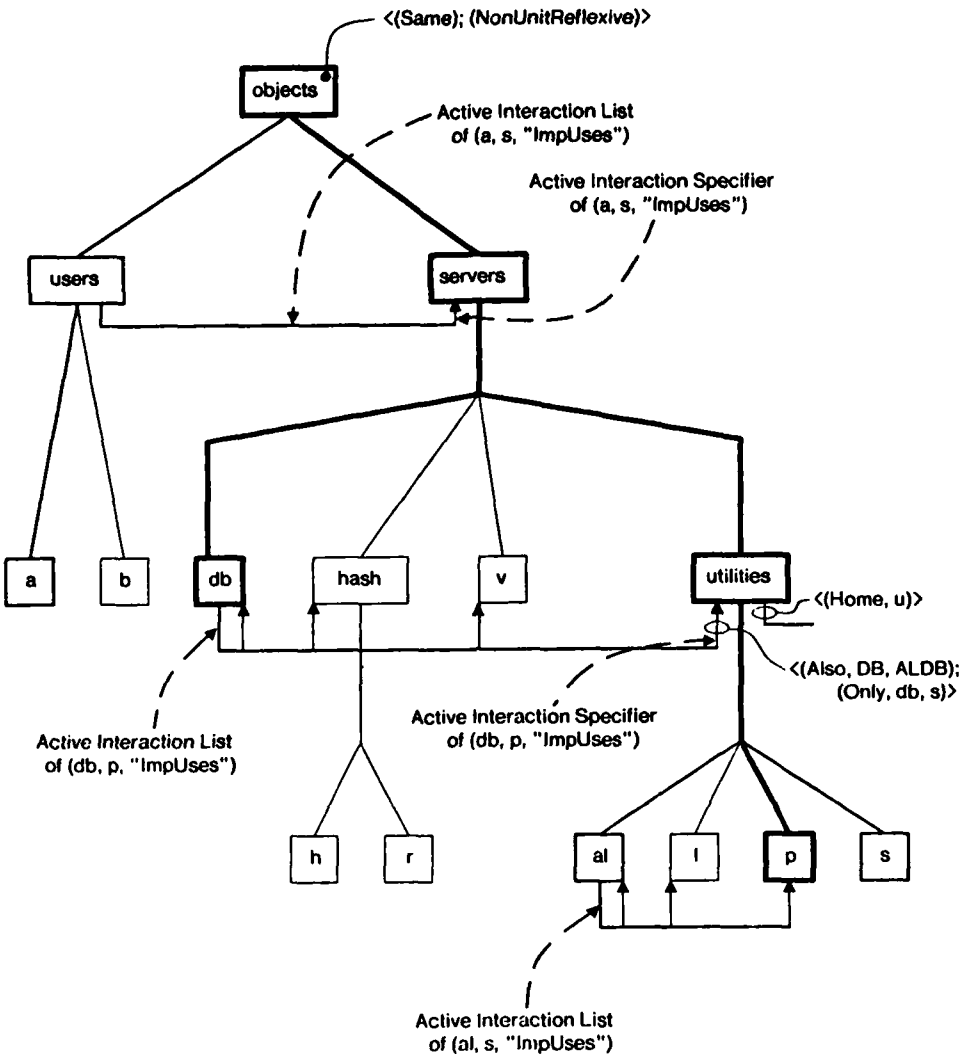


Figure 3-9: Active Interaction Lists and Specifiers

Example 3-23: The active interaction lists of (*db*, *p*, "*ImpUses*"), (*a*, *s*, "*ImpUses*") and (*al*, *s*, "*ImpUses*") are shown in Fig. 3-9. The active interaction list of (*a*, *b*, "*ImpUses*") does not exist.

Definition 3-22: Let (n_1, n_2, rid) be a leaf interaction triple, let (s_1, s_2) be the sibling ancestor pair of (n_1, n_2), and let *il* be the active interaction list of (n_1, n_2, rid), if it exists. Then $I(il)$ is the set of interaction specifiers associated with *il*. If $\exists\ is = (s, Qi) \in I(il)$ such that $s = s_2$, then *is* is called the *active interaction specifier* of

(n_1, n_2, *rid*). If *il* does not exist, or if no such *is* exists, then the active interaction specifier of (n_1, n_2, *rid*) does not exist.

Example 3-24: The active interaction specifiers of (*db*, *p*, "*ImpUses*") and (*a*, *s*, "*ImpUses*") are shown in Fig. 3-9. The active interactive specifiers of (*al*, *s*, "*ImpUses*") and (*a*, *b*, "*ImpUses*") do not exist.

3.4.2. The Semantics of Ideal Grids

Throughout this section, we assume that we are dealing with an ideal grid, and hence that all qualifier sequences are null. In this case, the directories are entirely independent: the object directory deals only with objects and the view directory deals only with views.

The semantics of an ideal grid are defined in terms of the validity predicates of its directories:

Definition 3-23: The *validity predicate* of a directory D is a predicate

$$valid_D: L_D \times L_D \times Rid \rightarrow Boolean$$

where $L_D \subset L$ is the set of leaf nodes of directory D. The predicate determines whether a given leaf interaction triple is valid for D.

The details of $valid_D$ are given below.

The independence of the directories motivates the following simple definition of the semantics of an ideal grid:

Definition 3-24: Let $GRID = (U, Rid, M, N, GD, VD, L, leaf)$ be an ideal grid as defined in Section 3.3, let (u_1, u_2, *rid*) be an interaction triple, and let (g_1, v_1) be the leaf pair of u_1 and (g_2, v_2) be the leaf pair of u_2 in *GRID*. Then

$$valid_{GRID}(u_1, u_2, rid) = valid_{GD}(g_1, g_2, rid) \wedge valid_{VD}(v_1, v_2, rid)$$

where $valid_{GD}$ and $valid_{VD}$ are the validity predicates of the object and view directories, respectively.

It remains to define the semantics of qualifier-free directories by specifying the details of the predicate $valid_D$. All the groundwork was laid by the discussion of active interaction specifiers, so the definition is simple:

Definition 3-25: Let D be a directory, and let (n_1, n_2, rid) be a leaf interaction triple in D. Then $valid_D(n_1, n_2, rid)$ is true if and only if the active interaction specifier of (n_1, n_2, rid) exists.

Example 3-25: Let *IGDB* be the ideal grid derived from *GDB* by omitting all qualifiers. The active interaction specifier of (*db*, *p*, "*ImpUses*") exists, as shown in Example 3-24, so

$$valid_{GD}(db, p, \text{"}ImpUses\text{"}) = true$$

Examination of the view directory reveals that the active interaction specifier of (*views.db*, *views.db*, "*ImpUses*") exists also, so

$$valid_{VD}(views.db, views.db, \text{"}ImpUses\text{"}) = true$$

As a result,

$$valid_{IGDB}(DB, PDB, \text{"}ImpUses\text{"})$$

$$= valid_{GD}(db, p, \text{"}ImpUses\text{"}) \wedge valid_{VD}(views.db, views.db, \text{"}ImpUses\text{"})$$

$$= true$$

The active interaction specifier of (*a*, *s*, "*ImpUses*") exists, as shown in Example 3-24, so

$$valid_{GD}(a, s, \text{"}ImpUses\text{"}) = true$$

However, the active interaction specifier of (*views.a*, *u*, "*ImpUses*") does not exist in the view directory, so

$$valid_{VD}(views.a, u, \text{"}ImpUses\text{"}) = false$$

As a result,

$$valid_{IGDB}(A, S, \text{"}ImpUses\text{"})$$

$$= valid_{GD}(a, s, \text{"}ImpUses\text{"}) \wedge valid_{VD}(views.a, u, \text{"}ImpUses\text{"})$$

$$= false$$

3.4.3. The Semantics of Grids with Qualifiers

Qualifiers complicate the semantics of the grid in two ways. First, and most obvious, they affect the validity predicates of the individual directories: the validity of a leaf interaction triple can no longer be determined solely be examining interaction specifiers. Second, they can violate the independence of the directories: a qualifier in one directory can refer to nodes in the other, or to specific units. A qualifier in one directory can even override the other directory entirely, by specifying that a particular unit interaction triple is valid irrespective of the information in the other directory.

The semantic definition of grids with qualifiers is based upon the semantic definition of ideal grids, but with the following modifications to take account of the complications mentioned above:

- Qualifier sequences within a directory are used to determine validity, in addition to interaction specifiers.
- The validity predicates of the directories are replaced by *validity functions* that return four-valued rather than Boolean results. The additional values allow one directory to override the other when necessary.
- The validity functions of the directories take *unit* interaction triples as arguments, rather than leaf interaction triples.

Validity functions of directories are defined as follows:

Definition 3-26: The *validity function* of a directory, D, is a function

$$valid_D: U \times U \times Rid \rightarrow \{\ Valid, ValidHere, NotValidHere, NotValid\ \}$$

that determines whether a given unit interaction triple is valid for D.

The results *Valid* and *NotValid* are absolute, and override the results of the other directory. The results *ValidHere* and *NotValidHere* indicate that the other directory must also be checked. Details of this function are given in Section 3.4.4.

The semantics of the grid are now defined in terms of the validity functions of the directories:

Definition 3-27: Let *GRID* be a grid, and let (u_1, u_2, rid) be an interaction triple. Then

$$valid_{GRID}(u_1, u_2, rid) =$$
$$[valid_{GD}(u_1, u_2, rid) = Valid] \vee$$
$$[valid_{VD}(u_1, u_2, rid) = Valid] \vee$$
$$[valid_{GD}(u_1, u_2, rid) = valid_{VD}(u_1, u_2, rid) = ValidHere]$$

where $valid_{GD}$ and $valid_{VD}$ are the validity functions of the object and view directories, respectively.[33]

[33] Minor variations in this definition can lead to interesting alternative semantics. For example, one directory can be prohibited from overriding the other, or the second directory can be ignored completely unless the first returns the result *ValidHere*. There are many other cases in the semantic definition where minor changes result in interesting variations, and one advantage of this definition, and of the implementation based upon it, is that such variations can be explored easily. This issue is discussed further in Chapter 6.

3.4.4. The Semantics of Directories with Qualifiers

In the presence of qualifiers, the semantics of a directory are defined in terms of qualifier sequences rather than just interaction specifiers. Validity functions are defined for qualifier sequences, as follows:

Definition 3-28: The *validity function* of a qualifier sequence, Q, is a function

$$valid_Q: U \times U \times Rid \rightarrow \{ Valid, ValidHere, NotValidHere, NotValid \}$$

that determines whether a given unit interaction triple is valid for Q.

Details of this function are given in Section 3.4.5.

The validity of an interaction triple for a directory depends on its validity for a single qualifier sequence within that directory, called the *active qualifier sequence*. This sequence is an extension of the notion of an active interaction specifier, and is defined as follows:

Definition 3-29: Let (n_1, n_2, rid) be a leaf interaction triple, and let the path of n_1 be $p_1 = n_1, p_2, \ldots, p_k$, where p_k is the directory root. Then the *active qualifier sequence* of (n_1, n_2, rid) is

$$Qi + Ql + GQl_1 + GQn_1 + GQl_2 + GQn_2 + \ldots + GQl_k + GQn_k$$

where "+" denotes sequence concatenation, and

- Qi is the local qualifier sequence of the active interaction specifier of (n_1, n_2, rid). If this active interaction specifier does not exist, then Qi is null.
- Ql is the local qualifier sequence of the active interaction list of (n_1, n_2, rid). If this active interaction list does not exist, then Ql is null.
- GQl_i is the global qualifier sequence of the *rid* interaction list associated with node p_i. More precisely, let $il = (r, I, Ql, GQl) \in IL(p_i)$ be the unique interaction list in

$IL(p_i)$ such that $r = rid$. Then $GQl_i = GQl$. If no such il exists, then GQl_i is null.

- $GQn_i = GQn(p_i)$ is the global qualifier sequence associated with node p_i.

Example 3-26: Consider Fig. 3-9, showing the active interaction lists and specifiers of some interaction triples in the object directory of *GDB*. The active qualifier sequence of (*db*, *p*, "*ImpUses*") is

⟨(*Also*, *DB*, *ALDB*); (*Only*, *db*, *s*); (*Same*); (*NonUnitReflexive*)⟩

The active qualifier sequence of (*a*, *s*, "*ImpUses*") is

⟨(*Same*); (*NonUnitReflexive*)⟩

The active qualifier sequences of (*al*, *s*, "*ImpUses*") and (*al*, *p*, "*ImpUses*") are both

⟨(*Home*, *u*); (*Same*); (*NonUnitReflexive*)⟩

The active qualifier sequence of (*views.db*, *views.db*, "*ImpUses*") in the view directory is null. Since the view directory contains no qualifiers, all active qualifier sequences there are null.

The concept of active qualifier sequences is extended to units as follows:

Definition 3-30: Let D be a directory, let (u_1, u_2, rid) be an interaction triple, and let $n_1 = leaf_D(u_1)$ and $n_2 = leaf_D(u_2)$ be the leaf nodes corresponding to u_1 and u_2 in D. Then the active qualifier sequence of (u_1, u_2, rid) in D is precisely the active qualifier sequence of (n_1, n_2, rid).

Example 3-27: The active qualifier sequence of (*DB*, *PDB*, "*ImpUses*") in the object directory is

⟨(*Also*, *DB*, *ALDB*); (*Only*, *db*, *s*); (*Same*); (*NonUnitReflexive*)⟩

and in the view directory is null. The active qualifier sequence of (*A*, *S*, "*ImpUses*") in the object directory is

⟨(*Same*); (*NonUnitReflexive*)⟩

and in the view directory is null.

The semantics of directories are now defined in terms of the semantics of qualifier sequences as follows:

Definition 3-31: Let D be a directory, let (u_1, u_2, rid) be an interaction triple, and let Q be the active qualifier sequence of (u_1, u_2, rid) in D. Then

$$valid_D(u_1, u_2, rid) = valid_Q(u_1, u_2, rid)$$

The semantics of qualifier sequences are described in the next section.

3.4.5. The Semantics of Qualifier Sequences

Whether or not a unit interaction triple (u_1, u_2, rid) is valid for a qualifier sequence often depends on whether u_2 is "exported" to u_1. The notion of exports in the context of the grid is formalised in the following definitions:

Definition 3-32: Let n be a directory node. Then n is *directly exported* if and only if $GQn(n)$ does not contain a *Hidden* qualifier.

Example 3-28: In *GDB*, all nodes except *r* and *hash* are directly exported.

A node that is hidden is "known" only in the subtree rooted at its parent. A node that is directly exported is "known" in the subtree rooted at its grandparent. It is also "known" wherever its parent is known:

Definition 3-33: Let n and a be directory nodes, such that a is an ancestor of n, and let the path from n to a be $p_1 = n, p_2, \ldots, p_k = a$. Then n *is exported by* a if and only if each $p_i\ (1 \leq i < k)$ is directly exported.

Note that a itself need not be directly exported.

Example 3-29: In *GDB*, *h* is exported by *hash* and *s* is exported by *servers*, but *h* is not exported by *servers* because its parent, *hash*, is not directly exported.

The definition of *exported* is extended to units in the following manner:

> **Definition 3-34:** Let D be a directory, let u_1 and u_2 be units, let $n_2 = leaf_D(u_2)$ be the leaf node corresponding to u_2 in D, and let (s_1, s_2) be the sibling ancestor pair of (u_1, u_2) in D. Then u_2 *is exported to* u_1 in D if and only if n_2 is exported by s_2.

Note that if a grid contains no *Hidden* qualifiers at all, then every unit is exported to every other unit. This does not mean that all interactions are valid, however: validity is also affected by interaction specifiers and other qualifiers.

> **Example 3-30:** In *GDB*, *h* is exported to *db* and *s* is exported to *a*, but *h* is not exported to *a*. Note that *h* is also exported to *v* and *utilities*, but these do not in fact interact with it, as no appropriate sibling interactions are specified.

The simplicity of the definitions of exports above derives from the simple nature of the *Hidden* qualifier. More complex qualifiers could be used to specify more elaborate export schemes. One obvious extension is to allow a node to control which of its exported descendants are to be exported further. Another is to allow nodes to be exported only to specific "users". Yet another is to allow separate specifications of exports for each kind of interaction (relation identifier). Qualifiers to specify these and other extensions are not currently part of the grid mechanism, but they could be added if they prove to be necessary.[34]

In addition to exports, the semantics of qualifier sequences depend on the semantics of individual qualifiers:

---

[34] Adding such qualifiers would probably require a more elaborate semantic framework than currently provided, such as *active target qualifier sequences* analogous to the active (source) qualifier sequences already present. This would also have the advantage of creating symmetry between import-style specifications and export-style specifications; at present the grid favours import-style specifications. This issue is discussed further in Chapter 6.

Definition 3-35: The validity function of a qualifier, q, is a function

$$valid_q: U \times U \rightarrow \{\ Valid,\ ValidHere,\ Inapplicable,\ NotValidHere,\ NotValid\ \}$$

that determines whether a given pair of units is valid for q.

Details of this function are given in Section 3.4.6. The additional result, *Inapplicable*, reflects the fact that not all qualifiers affect the validity of all interaction triples: a qualifier that returns the value *Inapplicable* for a particular interaction triple is transparent as far as that triple is concerned, and has no effect on its validity. The following definition is convenient:

Definition 3-36: If q is a qualifier and u_1 and u_2 are units, then q is *applicable* to (u_1, u_2) if and only if $valid_q(u_1, u_2) \neq Inapplicable$.

Unlike the validity function of a qualifier sequence, which applies to interaction triples, the validity function of a qualifier applies to pairs of units. The reason is that the relation identifier that forms the third component of an interaction triple is used to determine validity if no qualifiers are applicable, but does not affect the semantics of individual qualifiers.

The semantics of qualifier sequences are now defined in terms of the notion of exports and the validity functions of individual qualifiers, as follows:

Definition 3-37: Let Q be a qualifier sequence in directory D, and let $(u1, u2, rid)$ be a unit interaction triple. Then:

$valid_Q(u_1, u_2) =$

ValidHere	if Q is null, the active interaction specifier of (u_1, u_2, rid) exists, and u_2 is exported to u_1

NotValidHere	if Q is null, but either the active interaction specifier of (u_1, u_2, *rid*) does *not* exist or u_2 is *not* exported to u_1
$valid_{head(Q)}(u_1, u_2)$	if Q is not null, and $head(Q)$ is applicable to (u_1, u_2)
$valid_{tail(Q)}(u_1, u_2)$	otherwise

Considering this definition together with the definition of the semantics of directories (definition 3-31) yields the following rules for determining the validity of an interaction triple for a particular directory:

- If the active qualifier sequence is null, then the validity of the interaction triple depends on the active interaction specifier and on exports. In the absence of *Hidden* qualifiers, exports are unrestricted, and the interaction triple is *ValidHere* if the active interaction specifier exists, and *NotValidHere* if it does not.
- If the active qualifier sequence is not null, then the validity of the interaction triple depends on the first applicable qualifier in the sequence. If no qualifier in the sequence is applicable, validity depends on the active interaction specifier and on exports, as in the case of a null sequence.

Example 3-31: As mentioned in Example 3-27, the active qualifier sequence of (*DB, PDB,* "*ImpUses*") in the view directory is null. Its active interaction specifier exists, and there are no *Hidden* qualifiers in the view directory. Consequently,

$$valid_{VD}(DB, PDB, \text{"}ImpUses\text{"}) = ValidHere$$

The active qualifier sequence of (*A, S,* "*ImpUses*") in the view directory is also null, but its active interaction specifier does not exist. Consequently,

$$valid_{VD}(A, S, \text{"}ImpUses\text{"}) = NotValidHere$$

The active qualifier sequence of (*DB, PDB,* "*ImpUses*") in the object directory is

$$\langle (Also, DB, ALDB); (Only, db, s); (Same); (NonUnitReflexive) \rangle$$

The validity of this triple is therefore determined by the first

applicable qualifier in the sequence (the *Only* qualifier; see Section 3.4.6). If none of the qualifiers were applicable, it would be *ValidHere*, as its active qualifier sequence exists, *PDB* is exported to *DB* in the object directory.

In an ideal grid, all active qualifier sequences are null, and no *Hidden* qualifiers are present. Thus the validity of an interaction triple depends entirely on the existence of its active interaction specifier. These are precisely the semantics of ideal grids defined in Section 3.4.2.

3.4.6. The Semantics of Qualifiers

This section completes the semantic definition of the grid by defining the semantics of individual qualifiers. The definition is based on the abstract syntax of qualifiers (definition 3-16), and makes use of unit sets and unit set membership (definitions 3-14 and 3-15):

Definition 3-38: Let q be a qualifier. Then the validity function of q, $valid_q$, is defined as follows:

If $q = (Except, S_1, S_2)$ then

$$valid_q(u_1, u_2) = \begin{cases} NotValid & \text{if } u_1 \in S_1 \text{ and } u_2 \in S_2 \\ Inapplicable & \text{otherwise} \end{cases}$$

If $q = (Only, S_1, S_2)$ then

$$valid_q(u_1, u_2) = \begin{cases} NotValid & \text{if } u_1 \notin S_1 \text{ or } u_2 \notin S_2 \\ Inapplicable & \text{otherwise} \end{cases}$$

If $q = (AlsoHere, S_1, S_2)$ then

$valid_q(u_1, u_2) = ValidHere$ if $u_1 \in S_1$ and $u_2 \in S_2$[35]

$Inapplicable$ otherwise

If $q = (Also, S_1, S_2)$ then

$valid_q(u_1, u_2) = Valid$ if $u_1 \in S_1$ and $u_2 \in S_2$

$Inapplicable$ otherwise

If $q = (Same)$, the leaf pair of u_1 is (g_1, v_1) and the leaf pair of u_2 is (g_2, v_2), then

$valid_q(u_1, u_2) = NotValid$ if $g_1 \neq g_2$ and $v_1 \neq v_2$

$Inapplicable$ otherwise

If $q = (SameOther)$, the leaf pair of u_1 is (g_1, v_1) and the leaf pair of u_2 is (g_2, v_2), then

$valid_q(u_1, u_2) = NotValid$ if q is in the object directory and $v_1 \neq v_2$, or if q is in the view directory and $g_1 \neq g_2$

$Inapplicable$ otherwise

---

[35] According to this definition, the *AlsoHere* qualifier overrides all export control provided by *Hidden* qualifiers, because it does not check that u_2 is exported to u_1. This is appropriate in some cases. A variant that does check exports might be more appropriate in others, and can be added gracefully if needed.

If $q = (Home, l)$, the leaf pair of u_1 is (g_1, v_1) and the leaf pair of u_2 is (g_2, v_2), then

$$valid_q(u_1, u_2) = \begin{cases} NotValid & \text{if } q \text{ is in the object directory and } v_1 \neq l \wedge (g_1 \neq g_2 \vee v_2 \neq l), \text{ or} \\ & \text{if } q \text{ is in the view directory and } g_1 \neq l \wedge (v_1 \neq v_2 \vee g_2 \neq l) \\ Inapplicable & \text{otherwise} \end{cases}$$

If $q = (NonUnitReflexive)$, then

$$valid_q(u_1, u_2) = \begin{cases} NotValid & \text{if } u_1 = u_2 \\ Inapplicable & \text{otherwise} \end{cases}$$

If $q = (Hidden)$, then

$$valid_q(u_1, u_2) = Inapplicable$$

Hidden qualifiers are used only to determine exports, as described in Section 3.4.5, and are inapplicable in all other contexts.

Many other qualifiers and many semantic variations are possible. The semantic framework laid out above, involving validity functions, qualifier sequences and exports, was designed to facilitate addition and modification of qualifiers.[36] Some suggestions for additional qualifiers, and a discussion of the considerations involved in adding them, appear in Chapter 6.

Example 3-32: The following are some examples, drawn primarily from the active qualifier sequences appearing in previous examples:

$$valid_{(Also,\ DB,\ ALDB)}(DB, PDB) = Inapplicable$$

$$valid_{(Also,\ DB,\ ALDB)}(DB, ALDB) = Valid$$

---

[36] For example, the return value *NotValidHere* is not used by any existing qualifiers, but is provided nonetheless for completeness and future expansion.

$$valid_{(Only,\ db,\ s)}(DB,\ SDB) = Inapplicable$$

$$valid_{(Only,\ db,\ s)}(DB,\ PDB) = NotValid$$

$$valid_{(Same)}(DB,\ SDB) = Inapplicable$$

$$valid_{(Same)}(DB,\ S) = NotValid$$

$$valid_{(NonUnitReflexive)}(DB,\ SDB) = Inapplicable$$

$$valid_{(NonUnitReflexive)}(DB,\ DB) = NotValid$$

$$valid_{(Home,\ u)}(AL,\ P) = Inapplicable$$

$$valid_{(Home,\ u)}(AL,\ PB) = Inapplicable$$

$$valid_{(Home,\ u)}(ALB,\ AL) = Inapplicable$$

$$valid_{(Home,\ u)}(ALB,\ PB) = NotValid$$

3.4.7. Examples

This section integrates the examples that have been used throughout the semantic definition above. It consists of three examples. The first two complete the determination of the validity of the two interaction triples that were used in most of the preceding examples. The third presents a step-by-step account of how the validity of a third interaction triple is determined.

Example 3-33: The active qualifier sequences of (*DB*, *PDB*, "*ImpUses*") were determined in Example 3-27. They are:

$$Qg = \langle (Also,\ DB,\ ALDB);\ (Only,\ db,\ s);\ (Same);\ (NonUnitReflexive) \rangle$$

$$Qv = \langle \rangle$$

in the object and view directories, respectively. The next step in determining the validity of (*DB*, *PDB*, "*ImpUses*") is to determine its validity for these qualifier sequences.

In determining $valid_{Qg}(DB,\ PDB,\ \text{"}ImpUses\text{"})$, consider the

qualifiers in order. The first qualifier, (*Also, DB, ALDB*), is inapplicable to (*DB, PDB*) because *PDB* is not a member of the unit set denoted by *ALDB*, which contains *ALDB* alone. The second qualifier, (*Only, db, s*), is applicable to (*DB, PDB*):

$$valid_{(Only,\ db,\ s)}(DB, PDB) = NotValid$$

Thus

$$valid_{Qg}(DB, PDB, \text{"}ImpUses\text{"}) = NotValid$$

and so

$$valid_{GD}(DB, PDB, \text{"}ImpUses\text{"}) = NotValid$$

Next consider the validity of *Qv*. Since *Qv* is null, $valid_{Qv}(DB, PDB, \text{"}ImpUses\text{"})$ depends on whether the active interaction specifier of (*DB, PDB,* "*ImpUses*") exists in the view directory, and on whether *PDB* is exported to *DB*. The active interaction specifier does exist, and, since the view directory contains no *Hidden* qualifiers, *PDB* is exported to *DB*. Thus

$$valid_{Qv}(DB, PDB, \text{"}ImpUses\text{"}) = ValidHere$$

and so

$$valid_{VD}(DB, PDB, \text{"}ImpUses\text{"}) = ValidHere$$

Combining the results of the validity functions of the two directories as required by definition 3-27 yields the result

$$valid_{GDB}(DB, PDB, \text{"}ImpUses\text{"}) = false$$

This is consistent with the fact that no interaction *DB ImpUses PDB* occurs in the program (see Fig. 2-6).

Example 3-34: The active qualifier sequences of (*A, S,* "*ImpUses*") were determined in Example 3-27. They are:

$$Qg = \langle (Same); (NonUnitReflexive) \rangle$$

$$Qv = \langle \rangle$$

In determining $valid_{Qg}(A, S, \text{"}ImpUses\text{"})$, consider first the *Same* qualifier:

$$valid_{(Same)}(A, S) = NotValid$$

because A and S are not in the same object slice or the same view. Thus

$$valid_{Qg}(A, S, \text{"}ImpUses\text{"}) = NotValid$$

and so

$$valid_{GD}(A, S, \text{"}ImpUses\text{"}) = NotValid$$

Since Qv is null, and the active interaction specifier of (A, S, "*ImpUses*") does not exist,

$$valid_{VD}(A, S, \text{"}ImpUses\text{"}) = NotValidHere$$

The interaction triple is thus rejected by both directories, and

$$valid_{GDB}(A, S, \text{"}ImpUses\text{"}) = false$$

This is consistent with the fact that no interaction *A ImpUses S* occurs in the program (see Fig. 2-6).

Example 3-35: This example gives a step-by-step account of how to determine whether (V, L, "*DefUses*") is valid for *GDB*. We will examine its validity in the object directory first, then in the view directory, and then combine the results. Fig. 3-10 illustrates key points in the object directory, and Fig. 3-11 in the view directory.

The leaf nodes corresponding to V and L in the object directory are v and l, respectively. The lowest common ancestor of v and l is *servers*, and so their sibling ancestor pair is (v, *utilities*). The "*DefUses*" interaction list of v contains an interaction specifier with target *utilities*: these are the active interaction list and specifier of (V, L, "*DefUses*"). Since the active interaction list and specifier themselves have null qualifier sequences, the active qualifier list consists just of global qualifiers on the path from v to the root:

$$Qg = \langle (Same); (NonUnitReflexive) \rangle$$

We now examine each of these qualifiers in turn, in search of the first applicable one. Since both V and L belong to the same view, the qualifier (*Same*) is not applicable, and since $V \neq L$, the qualifier

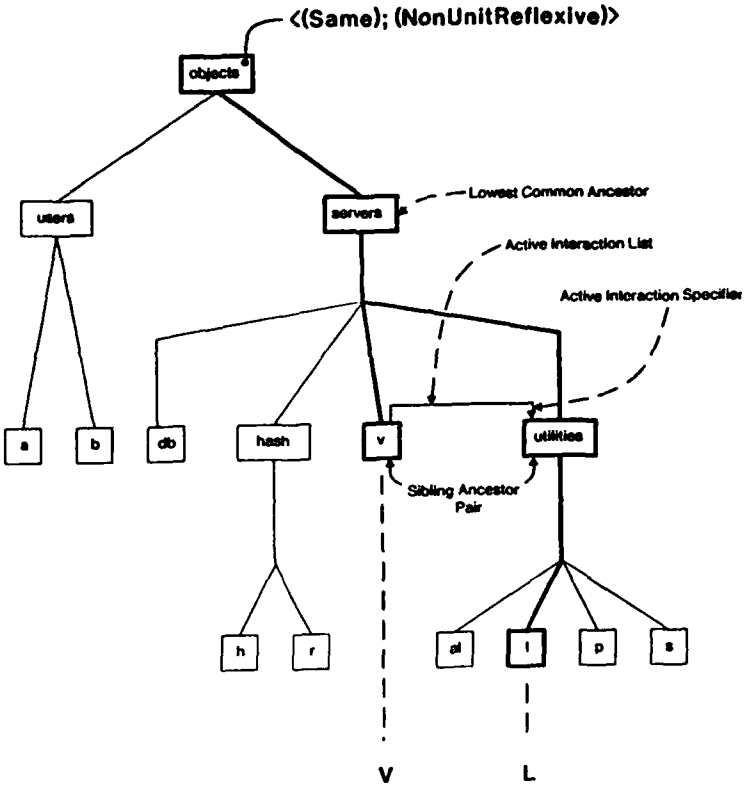


Figure 3-10: Testing (*V, L,* "*DefUses*"): The Object Directory

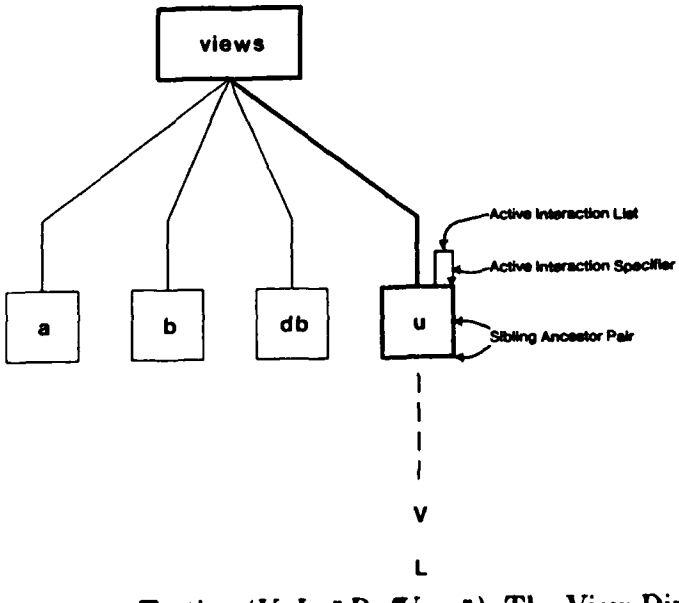


Figure 3-11: Testing (*V, L,* "*DefUses*"): The View Directory

(*NonUnitReflexive*) is not applicable either. The remaining qualifier sequence is null, so the validity of the interaction triple is determined by the existence of its active interaction specifier, and by exports. The active interaction specifier exists, as described above. Because *L* contains no hidden qualifier, it is directly exported. It is therefore exported by ***utilities***, and hence is exported to *V*. As a result,

$$valid_{Qg}(V, L, \text{"}DefUses\text{"}) = ValidHere$$

so

$$valid_{GD}(V, L, \text{"}DefUses\text{"}) = ValidHere$$

This process is now repeated in the view directory. The leaf node corresponding to both *V* and *L* in the view directory is *u*. By definition, their sibling ancestor pair is therefore (*u*, *u*). The "*DefUses*" interaction list of *u* contains an interaction specifier with target *u*, specifying the reflexive sibling interaction (*u*, *u*, "*DefUses*"): these are the active interaction list and specifier of (*V*, *L*, "*DefUses*"). The active qualifier sequence, *Qv*, of (*V*, *L*, "*DefUses*") is null, because there are no qualifiers in the view directory. Also, the active interaction specifier exists, and *u* is exported to *u*: a unit is always exported to itself, and, besides, there are no *Hidden* qualifiers in the view directory. Thus

$$valid_{Qv}(V, L, \text{"}DefUses\text{"}) = ValidHere$$

and so

$$valid_{VD}(V, L, \text{"}DefUses\text{"}) = ValidHere$$

Because

$$\begin{aligned} valid_{GD}(V, L, \text{"}DefUses\text{"}) &= valid_{VD}(V, L, \text{"}DefUses\text{"}) \\ &= ValidHere \end{aligned}$$

we conclude that

$$valid_{GDB}(V, L, \text{"}DefUses\text{"}) = true$$

This is consistent with the fact that the interaction *V DefUses L* does occur in the program (see Fig. 2-6).

The last example is rather long-winded, to show all the steps involved. In

practice, the validity of an interaction triple can often be determined at a glance from diagrams such as Figs. 3-10 and 3-11. This is especially true if few qualifiers are used.[37]

[37] Certain conventions in the use of qualifiers can help also. For example, if one never uses a qualifier in the view directory that can override the object directory by returning *Valid*, then one need examine the view directory only in the case of interaction triples that are *ValidHere* for the object directory. This convention can easily be enforced by a trivial change to definition 3-27, as is in fact done in the prototype implementation.

Chapter 4
The Prototype Implementation

A prototype implementation of the grid was constructed to test the semantics of the grid mechanism, and to facilitate future exploration of its features. Its internal structures closely mirror the abstract syntax defined in Section 3.3, and it implements the $valid_{GRID}$ predicate defined in Section 3.4 (with one minor variation, described below). As such, it can be used to test the validity of arbitrary interactions, and hence to check that the grid mechanism as defined behaves as expected and desired. It can also be used to explore variations in the semantic definition, and to try out new qualifiers. It was expressly designed and built with flexibility in mind, to make such exploration convenient.

The prototype implementation, as it stands, is *not* intended for general use, however. It is unsuitable for this purpose primarily for three reasons:

- It has an inadequate user interface. Grids are represented externally in a form that corresponds closely to the internal form, and no good facilities are provided for creating and manipulating them. The representation used has many advantages for testing, but is inconvenient for users.
- It is slow. Flexibility was considered paramount at this stage, and efficiency suffered as a result.
- It does not provide support for creating and manipulating multiple views of objects. The usefulness of the grid in practice largely depends on the extent to which programmers make use of multiple views of objects. As discussed in Section 2.1, they cannot be expected to do so without adequate programming environment support.

Nonetheless, the prototype implementation does provide a starting point for the construction of a more complete programming environment. Section 6.3 contains some suggestions for upgrading it and improving its efficiency.

The prototype implementation allows the grid mechanism to be used online to represent, document and enforce the structure of Modula-2 systems [Wirth 83]. In this context, a Modula-2 "system" means any collection of Modula-2 *definition*, *implementation* and *program* modules: it need not be a single program, and it need not be complete (i.e., modules within the system can refer to modules outside the system). The implementation consists of a collection of basic *tools* for manipulating grids, relations, and Modula-2 systems. These tools can be combined in traditional Unix[38] fashion [Ritchie-Thompson 74, Kernighan-Mashey 81] to perform a variety of useful, higher-level functions, such as comparing two grids or checking that all interactions occurring in a Modula-2 system are valid according to a grid. Only one of the tools actually deals with Modula-2 modules, and it is the only one that has any knowledge of the syntax or semantics of Modula-2; all the other tools treat units as atomic, and are completely language independent. In fact, the implementation was carefully constructed to facilitate portability to other languages.

This chapter describes the main features of the prototype implementation. Section 4.1 describes the overall structure of the implementation; some additional structural details are given in Section 5.2, which describes a grid specification of the implementation. The remaining sections discuss the tools provided, the implementation of the $valid_{GRID}$ predicate, and the intermediate form used for long-term storage of structures. The prototype implementation was used on three examples: the shared database example introduced in Chapter 2, the prototype implementation itself, and the Scribe document processing system.[39] These examples are discussed in Chapter 5, and extracts of computer input, output and intermediate structures associated with the shared database example are included as Appendix I.

---

[38] Unix is a trademark of AT&T Bell Laboratories.

[39] Scribe is written in Bliss, not Modula-2. Conversion of the prototype implementation to handle it proved simple, and is described in Section 5.3.

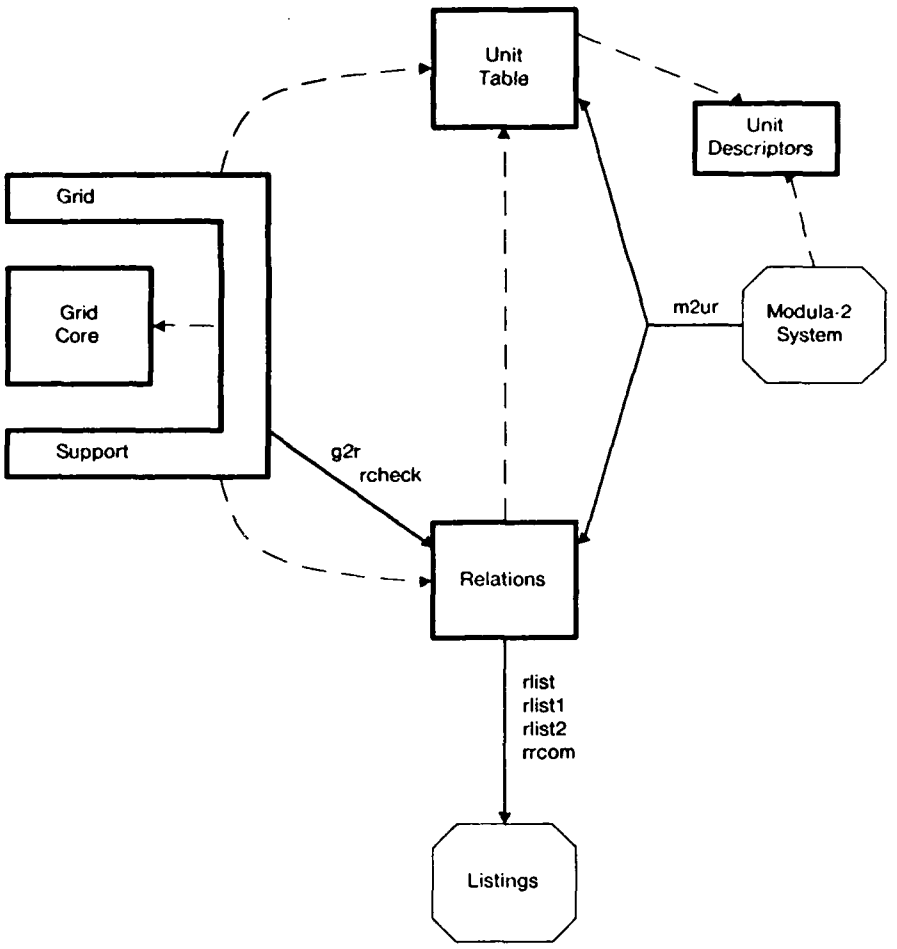


Figure 4-1: Overall Structure of the Prototype Implementation

4.1. Overall Structure

The overall structure of the prototype implementation is governed by the desire to keep the grid mechanism language-independent, yet to use it to specify the structure of actual systems written in some specific language, called the *unit language* (in this case, Modula-2). Three techniques are used to achieve these aims:

- All language-dependent information is encapsulated within two abstract data types and one tool. The internals of these can be changed with no disruption to the rest of the implementation.

- Two structures are introduced to serve as an interface between the grid and the unit language: *unit tables* and *relations*. A unit table characterises the units making up a system, and relations characterise the interactions between them. These structures are independent of both the grid and the unit language.
- The grid itself is separated into two parts: the *core*, which knows nothing of program- or programming language-related concepts at all, and surrounding *support*, which makes limited and restricted use of such concepts.[40]

The resulting structure is illustrated in Fig. 4-1.

The abstract data types representing language-dependent information are *source positions* and *unit descriptors*.[41] The tool that deals with Modula-2 programs is called *m2ur*, which stands for "Modula to unit table and relations". A source position specifies a location in a system, usually of a unit or an interaction. Source positions are used by various of the tools making up the prototype implementation to provide output that is helpful to the user. A unit descriptor contains language-dependent information about a unit. Unit descriptors are used only by *m2ur*, though they are stored in unit tables and are potentially available for use by any language-dependent software.

A unit table is an atom table that stores information about the units of a system. A separate unit table is constructed for each system, and it completely characterises the units of that system. One of the tasks of *m2ur* is to create the unit table corresponding to a system. A unit table stores the following information about each unit:

---

[40]This separation is conceptual rather than physical: core and support procedures sometimes appear in the same module. An analysis of structure reveals that it is valid to consider the core and support parts to be separate nonetheless, and that physical separation would be easy to achieve if desired.

[41]Source positions are not shown in the figure: they are encapsulated within a collection of utilities that are available for general use. The type identifiers used in the actual program code are "SourcePosition" and "ProgramInfo". In general, I have tried to avoid using program tokens in the text of this thesis; in most cases, the correspondence between text and code should be clear. An exception is tool names, which are used unchanged in the text.

- A unique *atom name*, assigned to the unit by the table.
- A *unit identifier*, derived from the program code by *m2ur*.
- A *unit descriptor*, also derived from the program code by *m2ur*.

Atom names are the sole means of referring to units internally throughout the implementation. Unit identifiers are used by various of the tools to provide output that is meaningful to the user. Unit descriptors are used only by *m2ur* itself.

A *relation* specifies interactions of a particular kind between units. A unique *relation identifier* associated with each relation denotes the kind of interactions it specifies. A relation uses atom names to refer to units, and is bound to a unit table that provides an interpretation of those names. In addition to recording interactions, a relation records the position in the source program at which each interaction is specified. The other task of *m2ur* is to create two relations, *DefUses* and *ImpUses*, describing the interactions within a Modula-2 system.

The *core* of the grid implements the $valid_{GRID}$ predicate, as defined in Chapter 3. It makes use of atom names and relation identifiers, but not of any of the language-dependent abstract data types, nor even of unit tables and relations. The surrounding *support* provides higher-level functions based on the $valid_{GRID}$ predicate, such as producing a relation describing all interactions of a given kind in the grid. It makes use of unit tables, relations and source positions, but does not have direct access to the representation of any language-dependent information.

Some additional structural details are given in Section 5.2.

4.2. Tools

The basic tools making up the implementation were shown in Fig. 4-1.[42] They are as follows:

m2ur Creates the unit table and two relations that characterise a Modula-2 system.

g2r Creates a relation specifying all the interactions of a particular kind that are valid according to a grid.

rcheck Checks that all the interactions specified by a relation are valid according to a grid, and produces error messages identifying all invalid interactions.

rlist Lists a relation. Each line of the listing consists of a source unit, a target unit, and the source position at which the interaction occurred.

rlist1 Lists a relation. Each line of the listing consists of a source unit and all the target units with which it interacts. Source positions are omitted.

rlist2 Lists a relation. Each line of the listing consists of a target unit and all the source units that interact with it. This is therefore an inverted listing. Source positions are omitted.

rrcom Compares two relations, identifying whether each interaction occurs in the first, the second or both.

These tools can be combined to form a variety of useful, higher-level tools, such as the following:

glist Lists all the interactions of a particular kind that are valid according to a grid.

mcheck Checks that all the interactions in a Modula-2 system are valid according to a grid, and produces error messages identifying all invalid interactions. This is the principal tool used for enforcing the structure specified by a grid.

gmcom Compares a grid and a Modula-2 system, identifying whether each interaction occurs in the grid, the system or both.

---

[42] The tools are "basic" from the point of view of this exposition; some of them consist of combinations of various programs, including several standard Unix programs.

ggcom Compares two grids, identifying whether each interaction occurs in the first, the second or both.

An interesting and useful by-product of the prototype implementation is the ability to use some of the tools, often in combination with Unix programs and editor functions, to analyse the structure of Modula-2 programs independent of the grid. For example, one can produce a "makefile" [Feldman 79] for a Modula-2 system, list the *DefUses* and *ImpUses* interactions in standard and inverted form, and explore different clusterings by selecting a collection of units and determining all interactions between units in the collection and units outside it. This usage has proven convenient in analysing the structure of the prototype implementation itself as part of the process of creating a grid to specify its structure. It is an illustration of how (in this case very rudimentary) structure analysis tools can be used successfully in parallel with the grid mechanism.

4.3. The Grid Core

The grid core is the part of the prototype implementation that implements the $valid_{GRID}$ predicate. Its internal structure is based on the abstract syntax of the grid defined in Section 3.3, though it differs in some details, primarily in the interests of efficiency and convenience. For example, instead of *Hidden* qualifiers being treated as global qualifiers associated with a directory node, they are implemented as a Boolean field "Exported" that is false if the node is hidden.

The semantics of the implementation are as defined in Section 3.4, with one exception: the validity of an interaction is first checked in the object directory, and only if the result is *ValidHere* is the view directory examined. This approach does not affect the semantics of the grid provided no qualifier in the view directory returns the result *Valid*, overriding the object directory. No such qualifier has yet appeared in any of the examples used.

It is worth noting that the implementation of the core does not explicitly

construct active qualifier sequences; it merely examines qualifiers in the appropriate order. A variety of other techniques are also used to improve efficiency without affecting semantics or flexibility, though there remains much scope for improvement.

The implementation of the core contains many hooks for anticipated future features, such as more or different qualifiers and direct support for approximation. Some of these are mentioned in Chapter 6.

4.4. Intermediate Form

Grids, relations and unit tables are used for communication between different tools. They are also used for communication between tools at different times, and they must generally remain in existence for as long as the Modula-2 systems they describe. To facilitate such communication and long-term storage, all structures are saved in a standard *intermediate form*, suitable for use by all tools at all times.

The intermediate form chosen is an ASCII encoding based on the IDL canonical external representation [Nestor-Wulf-Lamb 81]. This form treats all structures as collections of nodes with attributes; links between nodes are just examples of attributes. In the external form, each node is represented by a list of attribute-value pairs. Each attribute is named explicitly and in full. Values appear literally in some contexts, and as references in others.[43] Fig. 4-2 shows an extract of a saved relation in this form, and Fig. 4-3 shows a complete directory node (node *v* of the shared database example, illustrated in Fig. 3-3). It is clear

[43] IDL allows a value in any context to appear either literally or as a reference. This is the only significant difference between the grid intermediate form and the IDL external representation.

```
. . .
interaction [
        source          4;
        destination     9;
        position        SourcePosition [
                fileName        ./DBA.def;
                tokenNumber     6
                ]
        ]
interaction [
        source          4;
        destination     25;
        position        SourcePosition [
                fileName        ./DBA.def;
                tokenNumber     9
                ]
        ]
. . .
```

Figure 4-2: An Extract of a Relation in Intermediate Form

from the figures that the intermediate form is clumsy.[44] On the other hand, it is easy to read, to edit and to relate to the internal structures. As such, it is ideal for use during program development and testing, while the internal structures are in a state of flux.

A version control mechanism built into the prototype implementation allows changes to the intermediate form to be handled gracefully. Such changes occur frequently during program development, as they are often needed to reflect changes made to the internal structures. A separate version number is associated with each data type, and controls the manner in which values of that type are read and written. Only the latest version of a type is ever written, but code is retained to read earlier versions whenever possible. When a structure is saved in intermediate form, all the current version numbers are written at the beginning of the file, for use when the structure is loaded at a later stage.

---

[44] In fact, reading and writing the external form appears to be the major cause of inefficiency in the prototype implementation.

```
Node [
        Name            8;
        Id              v;
        Parent          3;
        FirstChild      0;
        NextSibling     9;
        Vector[45]      6;
        Exported        TRUE;
        Interactions    <
                InteractionList [
                        RelationId      DefUses;
                        Entries         <
                                Interaction [
                                        Sibling 9
                                        ]
                                >
                        ]
                InteractionList [
                        RelationId      ImpUses;
                        Entries         <
                                Interaction [
                                        Sibling 8
                                        ]
                                Interaction [
                                        Sibling 9
                                        ]
                                >;
                        GeneralQualifiers[46]    <
                            Qualifier [
                                Kind            Home;
                                HomeVector      4
                                ]
                            >
                        ]
                >
        ]
```

Figure 4-3: Directory Node v in Intermediate Form

---

[45] "Vector" is an obsolete term for "Slice".

[46] "General qualifier" is an obsolete term for "global qualifier".

The prototype implementation requires the grid specifying a Modula-2 system to be supplied by the user in intermediate form. This is clearly inconvenient, but is adequate for the purposes of demonstrating the functionality of the grid. One of the first enhancements to the implementation would be the addition of a parser to allow the user to supply grids in a more convenient form.

Chapter 5
Examples

This chapter discusses the use of the grid mechanism to specify the structure of three example programs: the shared database example introduced in Section 2.1.1, the grid implementation itself, described in Chapter 4, and the Scribe document processing system [Reid 80, Reid-Walker 80]. The programs differ greatly in nature and in size, and the last is an example of a real, large system that is in widespread use. Sections 5.1, 5.2 and 5.3 describe the three examples, and evaluate the effectiveness with which the grid can specify their structure.

An important question in evaluating the grid mechanism is whether it will scale well: will a grid specification grow unreasonably large and complex as program size grows, or will it remain a concise and readable specification of structure? Section 5.4 discusses this issue in the light of the three examples.

5.1. The Shared Database Example

The shared database example introduced in Section 2.1.1 has already been discussed extensively, and a complete grid specification of its structure was shown in Figs. 3-1 through 3-3. This section describes the use of the prototype implementation on this example, including an experiment with qualifiers. Representative extracts of input and output, and of structures in intermediate form, are included as Appendix I.

To allow the shared database example to be processed by the prototype implementation, a skeleton Modula-2 module, consisting just of *import*

specifications, was constructed for each definition and implementation subunit, and the grid specification was encoded in intermediate form. The tools were then used to generate the unit table and the *DefUses* and *ImpUses* relations characterising the program, to list the relations, and to check the relations against the grid specification.

The tool *mcheck* was run to check actual interactions in the program against the grid specification: it reported that all interactions are valid. There are 45 *ImpUses* interactions in the program. Checking these took took 15.2 processor seconds on a lightly loaded Vax 11/750.[47] Much of this time was spent reading the intermediate forms of the grid, unit table and *ImpUses* relation, which the prototype implementation does inefficiently. To determine the time actually taken by the grid core to check the interactions, a stub was substituted for the core, and the interactions "checked" again; this run took 12.1 seconds. The grid core therefore spent 3.1 seconds checking the 45 interactions, for a rate of approximately 14.5 interactions per second. This rate is rather slow, but is reasonable if one considers that few units, even in a large program, interact with more than 14 other units. The representation of the core lends itself to a variety of optimisation techniques that should improve the checking rate considerably; some of these are discussed in Section 6.3.

The tool *gmcom* was used to compare the grid and the program. It revealed that the grid specifies the structure of the program with complete accuracy: all interactions occurring in either the grid or the program occur in both. The full report produced by *gmcom* is included as Appendix I.7.

An experiment was performed to explore the effect of omitting qualifiers, and hence of the process of approximation. All qualifiers were stripped from the grid, and *gmcom* was run. Then the qualifiers were replaced, one kind at a time, and

---

[47] All timings are averages of a few runs, measured using the Unix timer.

Number of units:	27
Number of interactions tested:	729 (27^2)
Number of checks in the object directory:	729
Number of *DefUses* interactions in the program:	22
Number of *ImpUses* interactions in the program:	45

Table 5-1: Some Statistics of the Shared Database Example

Qualifiers	**Valid Interactions**	**View Checks**
None	31	124
Hidden	31	124
+ *NonUnitReflexive*	31	124
+ *Same*	31	31
+ *Home*	31	31
+ *Only* and *Also*	22	22

Table 5-2: The Effect of Qualifiers on *DefUses*

Qualifiers	**Valid Interactions**	**View Checks**
None	198	318
Hidden	191	311
+ *NonUnitReflexive*	168	288
+ *Same*	84	123
+ *Home*	53	72
+ *Only* and *Also*	45	63

Table 5-3: The Effect of Qualifiers on *ImpUses*

gmcom was run each time. The results are summarised in Tables 5-2 and 5-3; the information in Table 5-1 is useful in interpreting the results. The "Qualifiers" column of each table of results describes the kinds of qualifiers present in the grid. The "Valid Interactions" column lists the number of valid interactions specified by the grid with those qualifiers. The "View Checks" column lists the number of interactions that were found to be *ValidHere* by the object directory, and hence required the view directory to be checked. The *ImpUses* table shows the power of the *Same* and *Home* qualifiers, both in reducing the number of valid interactions and the number of view checks. The tables reveal that the *Only* and *Also* qualifiers eliminate nine *DefUses* interactions and eight *ImpUses* interactions. These are listed in Table 5-4. They are sufficiently unimportant in most contexts that these qualifiers could be omitted by the process of approximation.

DBA *DefUses* ALA	DBA *ImpUses* ALA
DBA *DefUses* LA	DBA *ImpUses* LA
DBA *DefUses* PA	DBA *ImpUses* PA
DBB *DefUses* ALB	DBB *ImpUses* ALDB
DBB *DefUses* LB	DBB *ImpUses* LB
DBB *DefUses* PB	DBB *ImpUses* PB
DB *DefUses* ALDB	
DB *DefUses* LDB	DB *ImpUses* LDB
DB *DefUses* PDB	DB *ImpUses* PDB

Table 5-4: Deviations Corrected by *Only* and *Also*

The grid mechanism successfully specifies the structure of this small, but important, example. In particular, it identifies the three views of *db* described in Section 2.1.1, and restricts access to them as required. This demonstrates that the grid can handle the important case of a shared resource being used in different ways by different users.

5.2. The Grid Implementation

The prototype implementation of the grid, described in Chapter 4, is a layered, object-oriented program, consisting of some 12000 lines of Modula-2 [Wirth 83] code. A complete grid specification of its structure was written, and the implementation itself was used to check actual interactions in the code against the specification. The process of constructing the specification was a valuable exercise in the use of the grid mechanism, and also brought to light a number of structural errors and peculiarities in the program.

This section gives a brief overview of the grid specification, and discusses a few interesting structural issues and how they are specified. The complete grid, in the intermediate form used by the prototype implementation, is available from me on request. Section 5.3 contains a more comprehensive description of an even larger grid specification.

5.2.1. The Specification

The prototype implementation of the grid mechanism is a Modula-2 system consisting 11 programs. The system contains 168 units, many of the which are shared by several programs. Each unit consists of a *definition module* and the *corresponding implementation* module, except:

- Each of the 11 main-program units consists of a single program module.
- Four Modula-2 library modules used by the implementation are treated as units, so that the manner in which they are used can be specified and documented. The composition of these units is not of interest.

The only kind of interaction specified in the grid is *Uses*: *a Uses b* if and only if the text of *a* (definition, implementation or program module) contains a reference to *b*.

The units in the system describe 67 objects from 10 points of view. The

matrix thus consists of 10 rows and 67 columns. Each column contains at most five units: no single object is described from more than five points of view. Many columns contain four units, and many others just one.

Four kinds of objects can be distinguished for convenience:

- *Programs*. Each main program is an object. These programs are closely related to the basic tools described in Section 4.2, but are not identical to them: some basic tools are made up of combinations of Modula-2 programs and standard Unix programs. There are 11 program objects in all.
- *Abstract data types*. The abstract data type discipline is used extensively throughout the system, and is enforced by means of the grid. Each abstract data type, however simple, is an object. There are 43 such objects.
- *Collections of procedures*. Nine objects consist of collections of procedures that are functionally related, but are not associated with any specific abstract data type.
- *Library Modules*. Each of the four Modula-2 library modules is an object.

The object directory specifies the hierarchical clustering of these objects, and their interactions. It consists of six levels. The top level merely segregates the objects into two clusters, ***users*** and ***utilities***, and specifies that the ***users*** can use the ***utilities***. Levels two and three, within the cluster ***users***, are the most interesting. They were described in Section 4.1 and illustrated in Fig. 4-1; details given there are referred to frequently below. Lower levels, and the details of ***utilities***, are not described in detail, though some points of special interest are discussed in Section 5.2.2.

Of the ten views, five are used extensively, primarily to enforce an extended variant of the abstract data type discipline:

- *Type*. Every abstract data type has a unit in this view, defining the type without specifying any of its properties: it consists just of an *opaque* type definition [Wirth 83].
- *Abstract*. This view specifies the operations provided by the abstract data type.

- *Public.* This view is an alternative to *Abstract*, used in the case of types and utilities that are considered public, such as strings. It specifies the operations provided by such a public type or utility.
- *Concrete.* This view specifies the representation of the abstract data type, in terms of other abstract data types or of types provided by Modula-2.
- *Save.* This view handles reading and writing of values of the type in intermediate form.

The other views serve special purposes, and are as follows:

- *Grid.* This view contains those *unit table* units that are available to the *grid core* or *grid support*. It is used to ensure that no language-dependent information in the unit table is accessible to the grid.
- *Read.* This view is related to *Save*, and deals with reading intermediate forms. It contains units that provide high-level primitives for performing the read operations.
- *Write.* This view analogous to *Read*, but provides write primitives.
- *Program.* This view contains all the units that are main programs.
- *Special.* This view contains some special units, exclusively for use by main programs. These units provide access to command-line parameters and the user's terminal.

The view directory specifies the hierarchical clustering of the views, and their interactions. The most important features are discussed in the next section.

5.2.2. Interesting Features

This section discusses some interesting features of the prototype implementation, the grid specification, and the process by which the grid specification was constructed.

5.2.2.1. The Abstract Data Type Discipline

The abstract data type discipline is enforced by the restriction that the *Concrete* view of an object can be used only by certain other views of the same object, but never by other objects. This restriction is specified by means of *Same* qualifiers in the view directory.

There are cases in the program, however, where an object does, in fact, use the *Concrete* views of other objects. This is a deliberate violation of the abstract data type discipline, permitted for reasons of convenience and/or efficiency. Most violations amount to a compromise that treats closely related objects as a cohesive group: units in such a group can use the *Concrete* views of certain other objects in the group, but no unit outside the group can use the *Concrete* view of any unit in the group.

Though such violations of the abstract data type discipline are quite common in programming, the ability to specify them is not. Languages that enforce a strict abstract data type discipline, such as CLU [Liskov, *et al.* 77, Liskov, *et al.* 81], do not allow violations, whereas languages that allow violations do not make the abstract data type discipline explicit, and do not distinguish between regular interactions and violations. In the grid specification, the strict abstract data type discipline is specified in the view directory, as described above, and violations are specified by means of *Also* qualifiers that override it. These qualifiers are in the object directory, attached to the nodes or interactions to which they apply. Each qualifier specifies a particular violation of the discipline, or a set of related violations, and serves to document and highlight them.

The question arises of why multiple views are used for handling abstract data types. The traditional approach in Modula-2 is to define the "abstract view" in a definition module, and then hide the "concrete view", together with implementation details, in the associated implementation module. The whole type is then defined within in a single unit, and multiple views arise only conceptually. The multiple view approach has two primary advantages over the traditional approach, however:

- More than one "abstract" view is possible. This is valuable for a variety of reasons, described below.
- Access to the concrete view by other types is possible. Such access should be used sparingly and be carefully controlled, but it is sometimes useful for reasons of convenience and efficiency.

Multiple "abstract" views of an abstract data type are useful for the following reasons:

- The operations provided by the type can be packaged in a variety of ways, based on function, intended users, or any other criteria. The separation of the *Abstract* and *Save* views is a good illustration of this.
- The operations made available to specific users can be restricted conveniently: different users can be given access to different views. The *Grid* view, for example, is used to restrict access to unit table operations, as described in Section 5.2.2.2.
- Recompilation can be reduced when changes are made. If a view is changed, only users of that specific view need be recompiled, rather than all users of the type. The separation of the *Abstract* and *Type* views is an illustration of this. Definition modules import only types, of which almost all are opaque and therefore defined in the *Type* view. If opaque types were defined in the *Abstract* view instead, any change to the abstract operation definitions of a type would require recompilation all units whose definition modules import that type.[48]
- Recompilation can be eliminated when new operations are added. If a new operation is needed, it can be placed in a view of its own, used only by units that need the operation. This entails no recompilation, whereas changing an existing view to include the new operation might require massive recompilation. Views introduced in this fashion should probably be regarded as temporary, to be removed as soon as the overhead of recompilation is acceptable. The *Read* and *Write* views are separate for this reason, as discussed in Section 5.2.2.6.

The implementation of each abstract view might require access to the concrete representation of the type. The traditional approach, which hides the concrete representation within a single module, therefore fails whenever there is more than one abstract view. This issue was the central theme of both the list example of Section 1.6 and the shared database example introduced in Section 2.1.1.

---

[48] Recompilation problems such as this could be eliminated by a sophisticated programming environment fine-grained enough to examine individual definitions. It is a fact of life, however, that many environments in widespread use do not address these problems. The multiple view approach is valuable in such cases. Actually, the *Type* view was originally introduced, not for this reason at all, but to overcome a more serious weakness of the Modula-2 environment: circular references among definition modules are not permitted.

5.2.2.2. Language-Independence of the Grid

An important design goal of the prototype implementation was to keep the actual grid (i.e. the *grid core* and *grid support* in Fig. 4-1) completely independent of the language in which the units are written. This was achieved primarily by the introduction of *unit tables* and *relations* to act as buffers between the grid and the unit language, as described in Section 4.1 and illustrated in Fig. 4-1. Relations do not contain any language-dependent information; the unit table does contain language-dependent *unit descriptors*, but hides them from the grid.

The above discipline is specified as follows:

- The grid is permitted to use unit tables and relations, but none of the objects that are language-dependent, such as unit descriptors. This is specified by means of sibling interactions.
- The grid is permitted to use only the *Grid*, *Type* and *Save* views of the unit table; the *Abstract* view is deliberately excluded. This restriction is specified by means of a single *Only* qualifier attached to the sibling interaction between the grid and the unit table. The permitted views contain no mention of the language-dependent information, and do not allow access to it.[49]

5.2.2.3. Restricted Use of Library Modules

The Modula-2 environment provides a number of library modules, of which four are used by the prototype implementation: *io*, *parameters*, *strings* and *unix*. Though these are publicly available from the point of view of Modula-2, only *io* is used freely within the prototype implementation; use of the others is restricted as follows:

- The library module *parameters* is responsible for providing access to command-line parameters. It is accessible only to main programs

---

[49] There are other ways of specifying this discipline. A particularly attractive way is to allow the grid to use only the *Grid*, *Type*, *Save* and *Public* views of *any* exterior object it is allowed to use at all. Then any non-public object that is to be used by the grid must provide a *Grid* view, explicitly specifying the operations made available to the grid. This approach is more elegant, but introduces some additional units.

(i.e. units in view *Program*). This is specified by placing *parameters* in the *Special* view, which is accessible only from the *Program* view.

- The prototype implementation contains abstract data types defining three different types of strings: variable-length, fixed-length and identifiers. These are publicly available to all other objects. The Modula-2 *strings* module, however, is used only to implement fixed-length strings, and is hidden from all other objects by means of clustering and a *Hidden* qualifier.
- The library module *unix* provides an interface to Unix system calls. It is used only by the error-handling object of the prototype implementation, and is hidden from all other objects.

Such restricted use of library modules is quite common, but the ability to specify it not. Some languages require library modules to be imported explicitly, others do not, but in neither case can a unit be prevented from importing and using a library unit. The grid mechanism provides several alternatives. A library unit can be:

- Hidden within a cluster, even though it is part of the language environment rather than the user program. This was done in the case of *strings* and *unix*.
- Placed in a view to which access is restricted, as in the case of *parameters*.
- Made universally available, eliminating the need for an explicit import specification in every unit that uses it, as in the case of *io*.
- Ignored completely, by treating it as part of the programming language rather than as a unit.

5.2.2.4. Saving of Intermediate Structures

The portion of the prototype implementation responsible for saving intermediate structures is interesting from a structural point of view. Each abstract data type that might have to be saved on an intermediate file has a unit in the *Save* view, specifying appropriate read and write operations. This unit has access to the *Abstract* and *Concrete* views of the same type; the *Concrete* view because it must know the representation, the *Abstract* view for convenience. The *Save* view of a type with components also has access to the

Save views of its components, so that it can cause them to be read or written. For convenience, it also has access to the *Abstract* views of its components (for operations such as *null* and *equal*), but never to their *Concrete* views.

Only three major kinds of objects are ever saved: grids, unit tables and relations. All other objects are saved only as components of these. Each of the three major types has a *Load* and a *Save* operation in its *Abstract* view, which invoke the read and write operations in its *Save* view. No other accesses to the *Save* view are permitted; this restriction is specified by means of a *Same* and an *Only* qualifier in the view directory. The *Save* view is thus an almost isolated layer of units, dedicated to performing a particular function.

The units in the *Save* view all use an object, called *Sio*, that performs the character-level input/output operations. Conversely, *Sio* is solely for the use of the *Save* view. This is an interesting case of an object being associated with a view. It is specified as follows:

- *Sio* is hidden in the object directory, and is the target of no sibling interactions. Thus it cannot be used, in the ordinary way, by any other object.
- A single *AlsoHere* qualifier in the object directory specifies that all units in the *Save* view can use *Sio*. The fact that the qualifier is *AlsoHere* rather than *Also* means that such interactions are still subject to the view restrictions in the view directory.

5.2.2.5. Multiple Programs

The prototype implementation consists of 11 programs. A few units are specific to individual programs, but most are shared by several programs. A single grid specifies the structure of all these programs, so information about the shared units need not be duplicated. The main-program units themselves are placed in a distinguished view, the *Program* view: a reader of the specification has only to scan this view to find all the programs.

This use of a single grid to specify the structure of a collection of related

programs suggests that the grid mechanism might be useful for specifying entire libraries of programs and/or modules.

5.2.2.6. Limiting Recompilation

The *Read* and *Write* views, providing high-level primitives for use by units in the *Save* view, might well be combined. The primary reason for their separation is that the write primitives were introduced after the read primitives were already in widespread use. Changing the *Read* view to incorporate write primitives would therefore have entailed a great deal of recompilation, whereas introducing the new *Write* view entailed none. This is an interesting use of views that is valuable in the short run, though views introduced this way should probably be treated as temporary and eliminated eventually.

In this particular case, the *Read* and *Write* views are implemented in terms of the *Abstract* view, and require no access to the *Concrete* view. In general, however, a new view of an abstract data type, created for whatever purpose, is likely to require access to the *Concrete* view. Such access is possible only if abstract data types are implemented in terms of separate views, as described in Section 5.2.2.1.

5.2.2.7. Revelation of Structural Violations and Peculiarities

An outline of the grid specification of the prototype implementation was written before programming began, and most aspects of the view structure were designed in advance. However, the detailed specification was constructed only after the program was complete. The process of constructing the specification and checking actual interactions in the code against it, revealed some violations of structural discipline and some interesting structural peculiarities. This is a valuable service provided by the grid.

A few of the violations and peculiarities were corrected in the program. In most cases, however, qualifiers were inserted into the grid specification to

account for them, since the program works well despite them. These qualifiers document the violations and peculiarities, and serve as reminders that they remain to be dealt with. This is another valuable service provided by the grid.

5.2.2.8. Grid Construction Aids

Since the grid specification was constructed largely after the program was written, some analysis of the program proved helpful in constructing it. Those tools in the prototype implementation that deal with relations and with Modula-2 systems were valuable in this regard. For example, *m2ur* was used to produce the *Uses* relation for the program, and various other tools were used to list the relation in convenient formats. More sophisticated analysis tools, especially tools based on the grid and able to assist with the exploration of alternative clusterings, would have been more useful still.

Another aid to grid construction was the use of qualifiers as *stubs*. When concentrating on one part of a grid, it is convenient to be able to ignore internal details of the rest simply by specifying that "anything goes". Details of interactions within a cluster, *c*, can be ignored by attaching the "stub" qualifier (*AlsoHere*, *c*, *c*) to the directory node representing *c*. For a considerable period of time, the entire view directory was ignored by attaching such a qualifier to its root. The use of "stub" qualifiers is discussed further in Section 5.3.3.4.

5.2.3. Checking Interactions

The prototype implementation itself was used to check the interactions actually present in its own code against the specification. There are 845 such interactions, all of which were reported as valid.

Checking the 845 interactions took 146.6 processor seconds on a lightly loaded Vax 11/750. Of this, the core took 66.3 seconds to perform the actual

checks, for a rate of approximately 13 interactions per second.[50] This rate is 90% of the rate measured in the case of the shared database example, described in the section 5.1, which is excellent considering the fact that the grid specification is considerably more complex in this case.

5.2.4. Evaluation

The grid implementation is a real, large program, written using the layered, object-oriented style. Its structure was designed with the grid in mind. In particular, multiple views of objects were used from the start, and proved to be an aid to program development, as well as to structure specification.

The grid mechanism successfully specifies the structure of the implementation, including a number of interesting structural features described above. The matrix identifies the objects and views explicitly. The object directory highlights similarities between views, by specifying interactions between objects independently of views (except for some qualifiers). The view directory specifies the abstract data type discipline, and other view restrictions, independently of specific objects. Qualifiers specify special cases and violations of the structuring discipline. This demonstrates the ability of the grid both to specify and enforce a strict structuring discipline, and yet to allow it to be overridden in a controlled and well-documented manner.

The grid is of manageable size and complexity, despite the size of the program; this issue is discussed in detail in section 5.4. In addition, information in the grid is well localised, so that only small chunks need be examined in any particular context. The implementation, though inefficient, is able to check interactions in its own code against the specification at an acceptable rate.

The grid implementation therefore serves as a first demonstration that the

---

[50] Determined using the stub technique described in Section 5.1.

grid mechanism can specify the structure of large programs in a concise and readable manner. An even larger example is described in the next section.

5.3. Scribe

The first example discussed in this chapter, the shared database example, is small, and though it illustrates a number of interesting points, it does not demonstrate that the grid is suitable for handling large programs. The second example, the prototype implementation, is a large program, consisting of some 12000 lines of Modula-2 code, and the grid mechanism succeeded in specifying its structure well. Another, even larger, example was desired, not written by me, to provide further evidence that the grid mechanism can handle large programs. The Scribe document processing system [Reid 78, Reid 80] was chosen, a grid specification of its structure was constructed, and the actual interactions occurring in the code were checked against the specification by a slightly modified version of the prototype implementation.

The Scribe source used as the basis of this example was the 1979 version numbered 2A(400) that was written by Reid at Carnegie Mellon University. It consists of approximately 29000 lines of Bliss [Wulf-Russell-Habermann 71] code. Reid himself provided much valuable assistance, explaining the structure to me, critiquing grid specifications, and helping to isolate the interactions actually present in the code.

This chapter describes the structure of Scribe by presenting the grid specification step by step. All diagrams used are derived directly from the grid: they are no more than graphical representations of portions of the directories. This exposition of the structure of Scribe in terms of the grid specification confirms the claim that the grid provides a suitable framework within which to describe and discuss the structure of large programs. The final sections of this chapter draw attention to interesting aspects of the specification, describe the

use of the prototype implementation to check the interactions in Scribe, and evaluate the performance of the grid on this example.

The description of structure that follows, and the accompanying diagrams, give full detail down to a certain level, as explained below, but no detail below that level. The full grid specification, in the intermediate form used by the prototype implementation, is available from me on request.

5.3.1. Units

The Scribe system consists of 11 *require files* and 43 *modules*. The require files list global names and define global macros; only such names and macros are accessible in more than one module. The modules themselves consist of data definitions, routines (procedures and functions), and local macros. Some routines contain nested routines or macros. The interactions of interest in this example are *references* of one unit by another, which are characterised by the relation *Uses*.

Three issues arise in the choice of units: *granularity*, *nesting* and *domain*. A coarse-grained specification of structure could be obtained by considering each require file and each module to be a unit, whereas an extremely fine-grained specification could be obtained by considering each definition within a require file or module to be a unit. A compromise was fact adopted, in which each individual definition is considered to be a unit, except that all the data definitions in a require file or module are considered to be a single unit. This keeps the number of units from becoming excessive, without reducing structural information significantly. Such flexibility in the choice of units is a useful advantage of the grid.

Since a unit is treated as atomic by the grid, one would expect constructs nested within units to be invisible. Too much structural information can be lost if this approach is followed, however, resulting in a structure specification that is

too coarse-grained. A finer-grained specification can be obtained by flattening the program, determining the units from the flattened program, and then specifying the nesting of units through the hierarchical structure of the object directory. This approach was in fact followed.[51]

The Scribe source contains a number of calls to operating system routines and machine operations. These can be treated as part of the Bliss language, and ignored, or they can be treated as units and calls to them specified explicitly in the grid. Since such calls are system-dependent, it is important to document them and the disciplined manner in which they are used. To this end, system routines and machine operations are treated as units.

The above considerations lead to a structure specification in terms of the following kinds of units:

Assembly.	A routine coded in assembly language for maximum efficiency.
DataDefs.	All the data definitions in a single module.
GlobalVar.	A variable that is available to more than one module.
Machop.	A machine operation invoked from Bliss code.
Macro.	A Bliss macro, including a macro nested within a routine.
Main.	The main program.
Proc.	A Bliss routine, including a routine nested within another routine.
Require.	All the data definitions in a single require file.
System.	A routine that is part of the operating system or language environment and is invoked by Scribe.

There are 1154 units in all.

---

[51] The low levels of the object directory were not developed in sufficient detail to show the nesting, however. All units, nested or not, that occur in a module are simply siblings in a cluster corresponding to that module, as described in Section 5.3.3.4. Specifying the nested structure within modules would be simple, if tedious.

The prototype implementation of the grid mechanism requires unit identifiers to be unique: it does not explicitly handle nested units or qualified names. Unique unit identifiers were derived as follows:

- The *DataDefs* unit associated with module *m* has identifier *m*[*data*].
- The *Require* unit associated with require file *r* has identifier *r*[*req*].
- The main program has identifier *scribe*[*main*].
- All other units have identifiers identical to those in the Bliss program, except that a disambiguating suffix is added when necessary. The suffix is usually of the form "[*module-name*]", but in the case of units local to different procedures within the same module it is "[*procedure-name*]".

5.3.2. The Matrix

Multiple descriptions of single objects do not occur in Scribe; there are no cases where two different units can be considered to specify a single object from different points of view or at different levels of abstraction. Instead, each unit is treated as a separate object.[52] There are thus 1154 object slices in the matrix, each containing but a single unit.

Despite the trivial nature of the object slices, views are nonetheless valuable as a means of categorising units according to kind and/or usage, as described in Section 5.3.4. Twenty-five categories of units proved useful, so there are 25 view slices in the matrix.

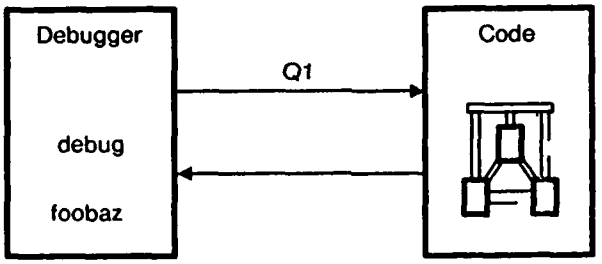
The matrix thus consists of 25 rows and 1154 columns, with exactly one unit in each column.

---

[52] It is convenient to use object terminology for consistency with the grid, even though Scribe would not normally be considered an object-oriented program.

5.3.3. The Object Directory

The object directory contains 1154 leaf nodes, one for each object in the Scribe system. The next level, called the *module level*, consists of clusters representing the require files and modules in the source program; the object directory thus preserves the modular structure of the source program (except for two important special cases discussed in Sections 5.3.6.3 and 5.3.6.4). Higher levels of the object directory specify further logical grouping not explicitly identified in the source program. The entire object directory is now presented top-down.

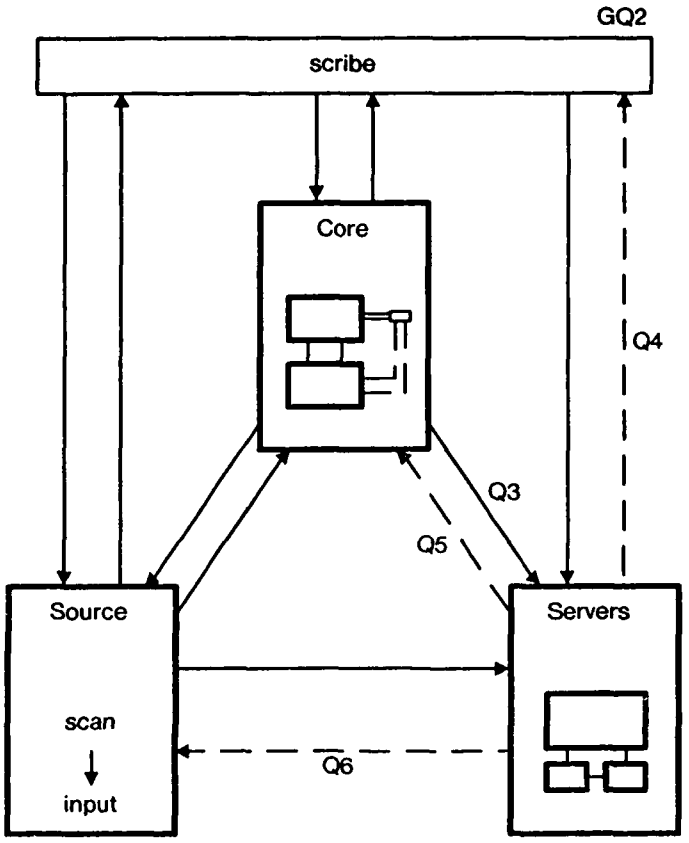


Q1: {(Also, Debugger, Debug[view])}

Figure 5-1: The Object Directory: Top Level

5.3.3.1. High Levels

At the highest level, the Scribe system consists of a debugger and code, as illustrated in Fig. 5-1. In this and subsequent diagrams, clusters representing modules are denoted by lower-case names, possibly enclosed in light boxes; in this case there are two: *debug* and *foobaz*. Higher-level clusters are denoted by heavy boxes with capitalised names; in this case there are two, *Debugger* and *Code*. A simple cluster might have its contents shown in detail, as in the case of *Debugger*; a complex cluster will instead contain an icon denoting a detailed diagram to follow, as in the case of *Code*.



GQ2: ⟨(Also, scribe, All-Scribe)⟩

Q3: ⟨(Also, dvdbl, ZWRString[DVDbl]);
(Also, dvxgp, ZWRString[DVXGP])⟩

Q4: ⟨(Only, {msg, osif10, symbol}, scribe)⟩

Q5: ⟨(AlsoHere, msg, driver);
(Also, msg, Send);
(Only, DecToJulian, DayNum)⟩

Q6: ⟨(Only, {msg, strn}, input)⟩

Figure 5-2: The *Code* Cluster: Top Level

The debugger was based on the standard Bliss debugger, but was hand-crafted to allow examination of important Scribe structures. Calls to the debugger are permitted within the code. The debugger itself can access all objects exported by *Code*, as well as some additional ones specified by the

qualifier *Q1*. All qualifiers in the grid are shown in the figures in which they belong, but are explained in Section 5.3.5.

The debugger is simple from a structural point of view, and does not merit further attention. The *Code* cluster consists for four sub-clusters, as shown in Fig. 5-2:

- *scribe* is the module containing the main program itself.
- *Core* is the heart of the Scribe compiler, consisting of all the routines for processing text and commands and producing finished output.
- *Source* consists of routines for reading and scanning input.
- *Servers* contains utilities for general use throughout the system.

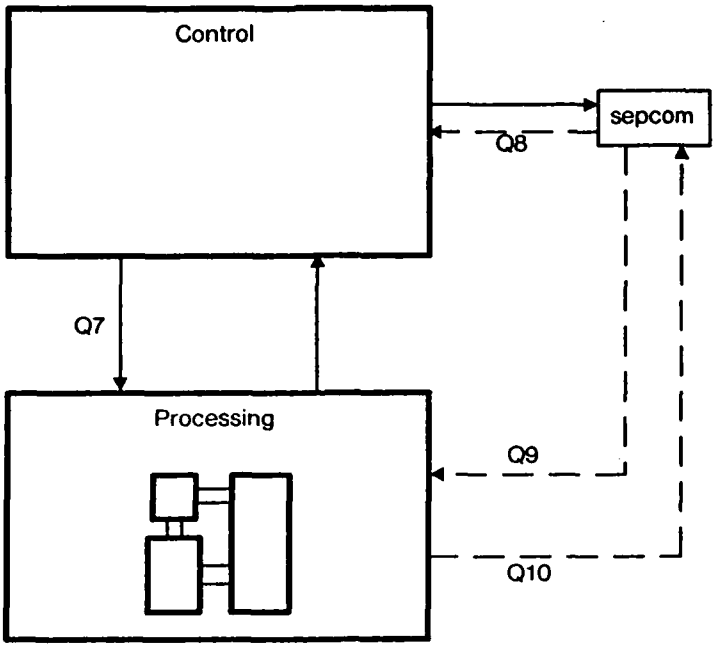
Solid arrows in all diagrams represent "normal" interactions; if they have qualifiers beside them, the qualifiers specify additional, extraordinary access (such as access to hidden units). Dashed arrows represent "restricted" interactions; they are always accompanied by qualifiers specifying the restrictions. The qualifiers are explained in Section 5.3.5. At this, high level it is sufficient to note that the major parts of the system can interact freely, except that *Servers* have only restricted access to the rest of the system. This is as expected, for servers are generally universally available, but largely independent.

Cluster *Source* is too simple to warrant further examination. Clusters *Core* and *Servers* are described in detail in Sections 5.3.3.2 and 5.3.3.3.

5.3.3.2. The Core

The top-level structure of *Core* is shown in Fig. 5-3. The three sub-clusters are as follows:

- *Control* contains modules that control the flow of data, examining input to distinguish between text and commands and causing it to be processed as appropriate.
- *Processing* contains the modules that actually convert input text to finished output, and maintain all the information required to do this.
- The module *sepcom* handles the separate compilation feature of Scribe, that allows parts of a document to be processed in isolation.



Q7: ⟨(Also, TextInit, {LINEcreate, HZFcode, ChrWid}); (Also, CMauxfont, DevFont)⟩

Q8: ⟨(Also, DefPart, WHERECMD); (Only, UsePart, Phase)⟩

Q9: ⟨(Also, DefPart, {BoxTest, PageBox, DevCapas, HCURSOR});
(Also, UsePart, {BoxAdd, LINEcreate, PageBox, CVDist, CtrSet});
(Also, PartClose, {DEVmark, Send, QUOTE, XRFMark});
(Only, {}, {})⟩

Q10: ⟨(Also, NOTINI, CtrRegister); (Only, {}, {})⟩

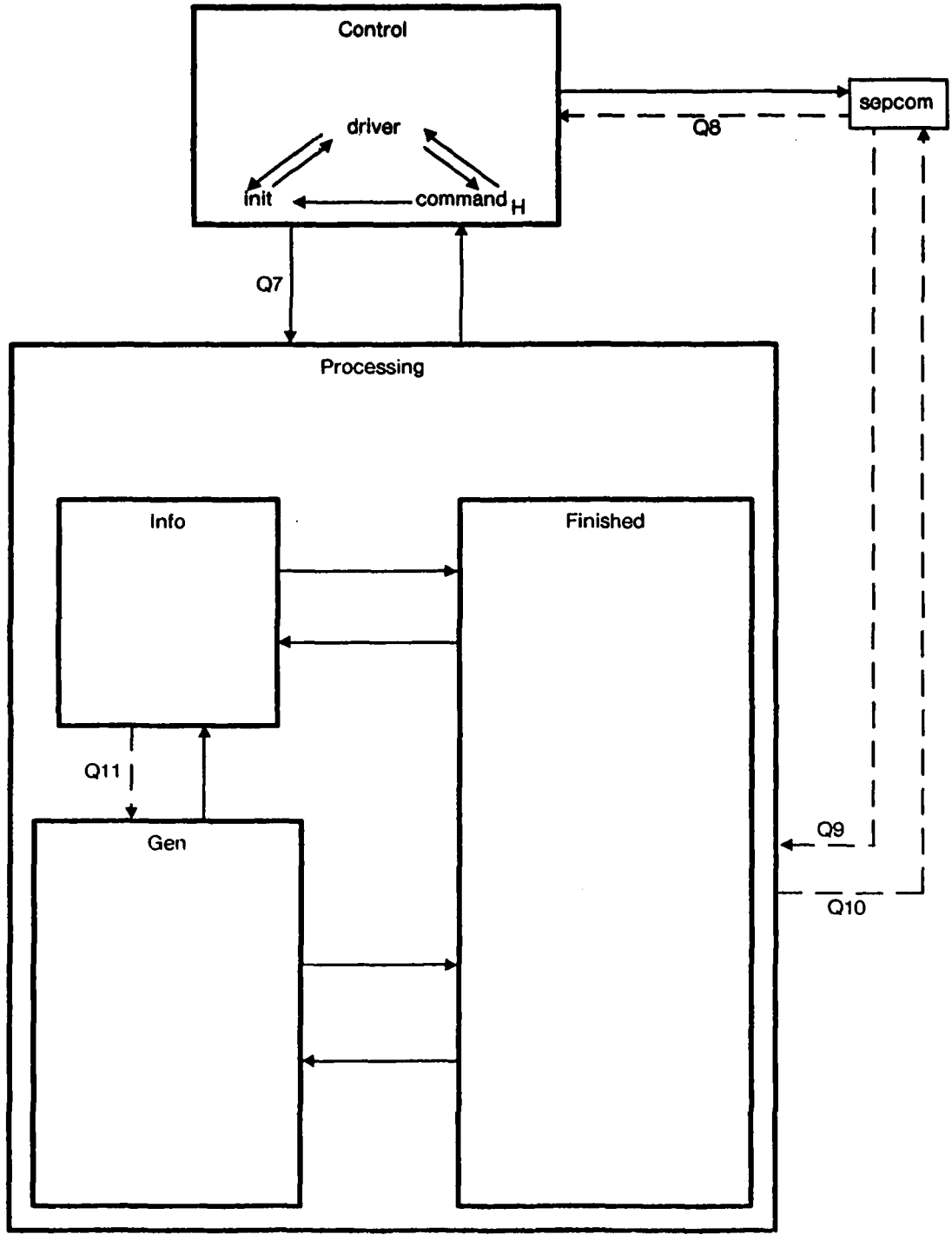
Figure 5-3: The *Core* Cluster: Top Level

> It is separate from the other clusters, and most interactions with it are restricted, because it is somewhat anomalous and violates information hiding constraints.

A further level of detail within the *Core* cluster is shown in Fig. 5-4.[53] *Control* is simple, consisting of just three modules:

---

[53] Because the core is a closely-knit portion of the program, and is of manageable size, it is helpful to introduce lower level detail while preserving the higher-level context, rather than showing details of sub-clusters separately.



Q11: ((Also, REFevaluate, XRref); (Also, AuxStyle, Send); (Only, {}, {}))

Figure 5-4: The *Core* Cluster: Second Level

- The module *driver* contains the main loop of Scribe, reading input, determining whether it is command or text, and invoking the appropriate unit to act upon it.
- The module *init* is responsible for *document initialisation*. This involves processing all the initial commands that occur before any text in the document, and includes loading of appropriate database files.
- The module *command* is responsible for executing all commands that occur in the document text.

The module *command* is hidden, as indicated by the "H" in the diagram: it is accessible only to *driver*.

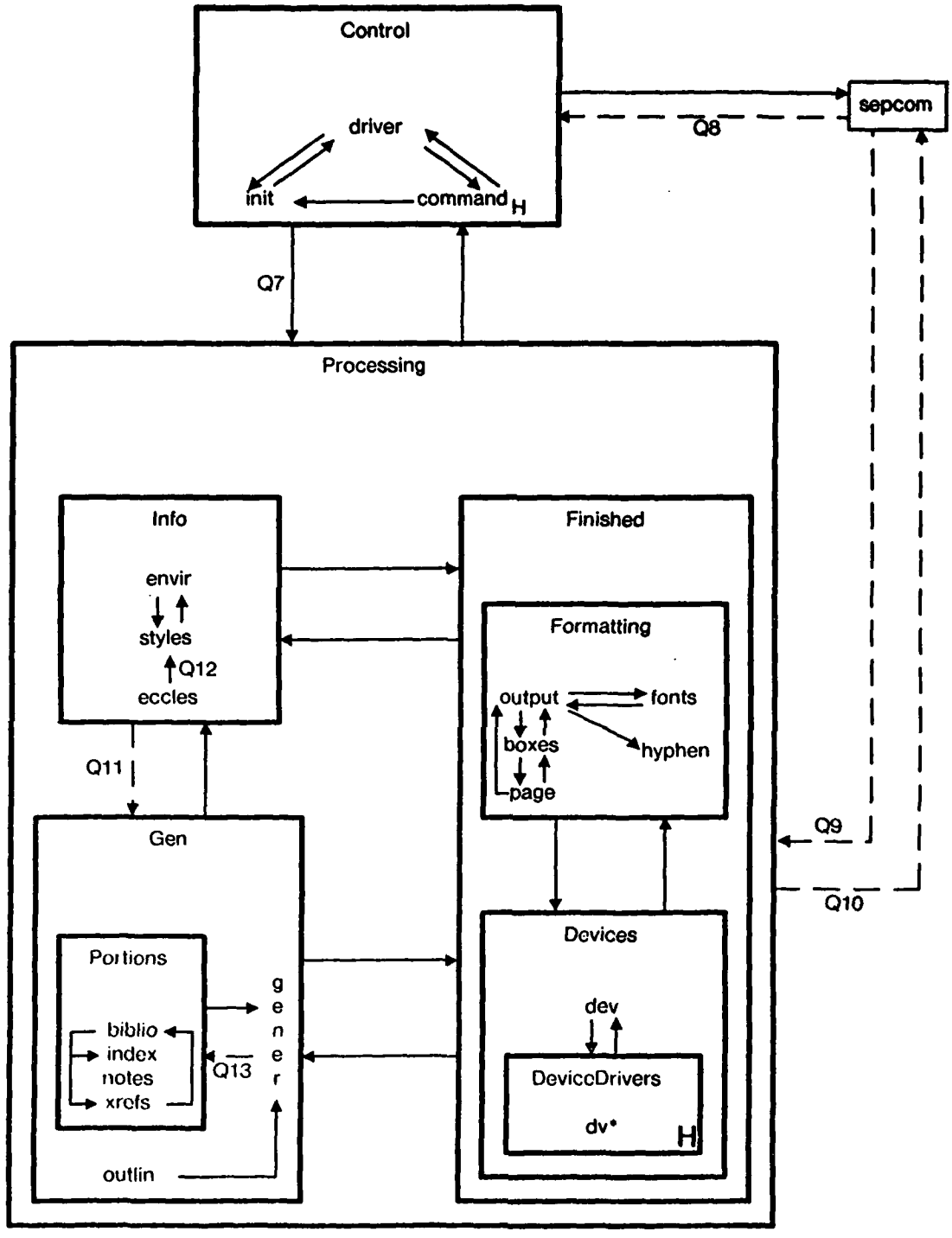
The *Processing* cluster is more complex, consisting of three sub-clusters:

- *Info* contains modules that maintain information about the document style and current environment, and make it available to other modules.
- *Gen* contains modules that are responsible for constructing the *generated portions*: those parts of the finished output, such as bibliography and index, that are constructed by Scribe rather than being obtained directly from the manuscript text.
- *Finished* contains the modules that actually synthesise and output the finished document.

Fig. 5-5 shows the *Core* cluster in full detail, down to the module level. Only clusters *Gen* and *Finished* are of interest. The sub-clusters within *Gen* are as follows:

- *Portions* contains modules to process the four primary generated portions: the bibliography, the index, foot and end notes, and cross-references.
- The module *outlin* produces the outline file; since this is an auxiliary file rather than part of the finished output, it is not considered to be a true generated portion.
- The module *gener* is responsible for actually having the generated portions included in the output.

The *Finished* cluster is of primary interest. At the first level, it consists of two sub-clusters:



Q12: ⟨(AlsoHere, Getevnt, ApEngOrdinal)⟩

Q13: ⟨(Only, gener, {IxActive, IndexClose})⟩

Figure 5-5: The *Core* Cluster: Full Detail

- *Formatting* contains modules that take text, whether from the manuscript file or generated, and format it for output. They deal with such issues as line breaking, justification, spacing, hyphenation, font selection, pagination and figure placement. All the modules are absolutely device-independent.
- *Devices* contains modules that produce the final raw output for the appropriate device, and provide details of device capabilities.

The *Devices* cluster is in turn divided:

- The cluster *DeviceDrivers* contains one module for each device supported by Scribe; there are five in this version. Each module produces raw output for its device. These are the only device-dependent modules in the entire Scribe system. The large "H" in the figure indicates that they are hidden within *Devices*.
- The module *dev* acts as an interface to the device drivers, exporting device-independent operations that it realises by calling the appropriate drivers.

The structure of *Devices* is such that outside modules can communicate only with module *dev*, ensuring that no device dependencies are introduced into the rest of the system.

5.3.3.3. Servers

The cluster *Servers* of Fig. 5-2 is expanded in Fig. 5-6. It consists of three sub-clusters:

- *High* contains high-level support, such as a system-independent operating system interface, some utilities, and configuration information.
- *Low* contains system-independent low-level support, such as macros for performing arithmetic or transfer operations efficiently.
- *Machine* contains all the machine operations that are used by Scribe.

Fig. 5-7 shows the *Servers* cluster in full detail, down to the module level (except that the fine structure and interactions within *BlissLibrary* are omitted from the diagram, though they are specified fully in the grid). The *BlissLibrary*, containing the Bliss input/output and memory allocation modules, is hidden, and is accessible to users only through the operating system interface, *osif*10.

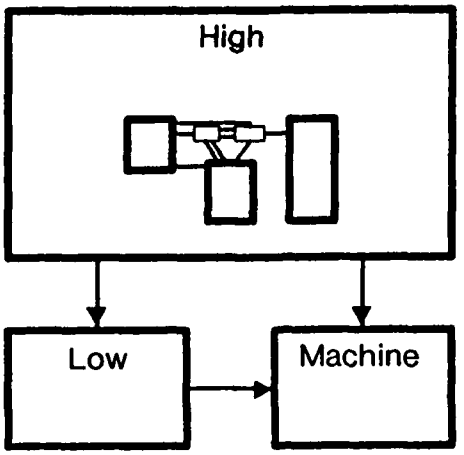


Figure 5-6: The *Servers* Cluster: Top Level

5.3.3.4. Low Levels

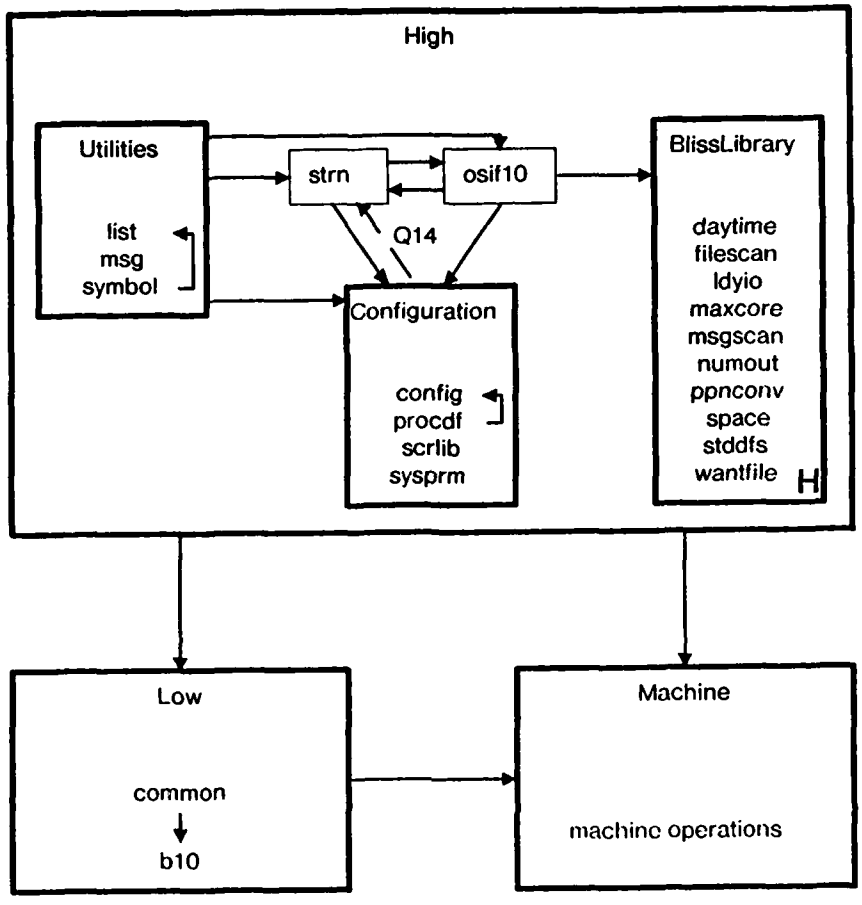
The previous three sections covered the entire object directory down to the module level. The qualifiers, described further in Section 5.3.5, and the views described in Section 5.3.4, often deal with specific units within a module; thus, although all sibling interactions discussed so far specify interactions between entire modules, or clusters of modules, the grid as a whole is finer-grained.

Nested structure within a module could be specified by means of additional levels of the object directory below the module level, and intra-modular interactions could be specified by means of sibling interactions between nodes at these levels. This detail would contribute little to an overall understanding of Scribe, and is not needed to demonstrate the ability of the grid to handle large programs. The following approach was therefore followed instead:

- All units in a module, whether nested within routines or not, are simply children of the cluster corresponding to the module.
- Each cluster corresponding to a module, *m*, has attached to it the global qualifier

  (*AlsoHere*, *m*, *m*)

  specifying concisely that all intra-modular interactions are permitted



Q14: ⟨(Only, config, StrCon)⟩

Figure 5-7: The *Servers* Cluster: Full Detail

by the object directory.[54] These qualifiers are not shown in the diagrams.

This approach results in an approximation that is appropriate in the present context, by omitting much low-level detail. The ability of the grid to specify

[54] It would be still better to make the notion of "module" explicit in the grid, perhaps by tagging nodes corresponding to modules. Then a single qualifier could be used to specify that all intra-modular interactions are permitted, instead of a separate qualifier being needed for each module. These facilities are not currently supported by the grid; tagging of nodes and the introduction of additional qualifiers are discussed in Section 6.1.

greater or lesser detail, and to specify different portions of the program at different levels of detail, is important. It is particularly valuable during grid construction, allowing a top-down approach to be used: qualifiers such as the above can be used as *stubs* that take care of portions of the program that have not yet been dealt with in detail.

5.3.4. The View Directory

The previous section showed how the objects making up the Scribe system are classified hierarchically according to function, in the object directory. It is also convenient to classify units according to kind and usage, so as to document system dependencies, restrict interactions, and handle some special structural issues. Views are used to achieve this second classification, and the views themselves are classified hierarchically in the view directory. This directory is now described top-down.

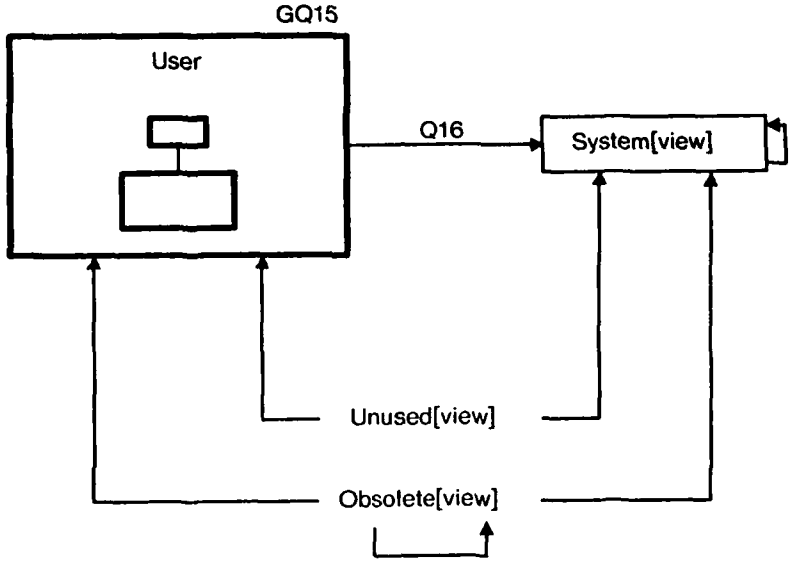
The top level of the view directory is shown in Fig. 5-8, and consists of:

- The cluster *User*, containing all units that are part of the Scribe system proper.
- The view *System*[*view*],[55] containing all machine operations and operating system routines. This view clearly identifies all units that form part of the particular computer system on which this version of Scribe runs. All units that use units in this view would have to be rewritten if Scribe were ported to a different system.
- The view *Unused*[*view*], containing all units that are not used at all. This view clearly identifies all unused units, so that they can be eliminated if desired.
- The view *Obsolete*[*view*], containing units that should be discarded. No other view uses this view, so any remaining interactions with an obsolete unit will be trapped by the view directory.

Units in *Unused*[*view*] and *Obsolete*[*view*] are allowed to interact freely with other views; since these units are likely to be discarded or ignored, what exactly

---

[55]The suffix "[*view*]" is appended to all view names to distinguish them from object and object cluster names.



GQ15: ((Only, User, User);
(AlsoHere, Debugger, Debugger); (AlsoHere, Control, Control);
(AlsoHere, Info, Info); (AlsoHere, Source, Source);
(AlsoHere, Gen, Gen); (AlsoHere, Finished, Finished);
(AlsoHere, sepcom, sepcom); (AlsoHere, scribe, scribe);
(AlsoHere, Servers, Servers); (AlsoHere, Gen, Info);

(Except, Device[view], Restricted);
(AlsoHere, {Control, Info, Gen, scribe}, All-Command);
(AlsoHere, Control, All-Source);
(AlsoHere, {biblio, styles, strn, Finished}, Consume[view]);
(AlsoHere, INIT, DocInit);
(AlsoHere, envir, Envir[view]);
(AlsoHere, Finished, {Finished[view], Joint[view]});
(AlsoHere, {styles, msg}, Get[view]);
(AlsoHere, Processing, Processing[view]);
(AlsoHere, {envir, Gen, Finished}, Produce[view]);
(AlsoHere, sepcom, SepCom[view]);
(AlsoHere, {Gen, msg}, Set[view]);
(AlsoHere, Source, Source[view]))

Q16: ((AlsoHere, Code, SIX12);
(AlsoHere, {SysInt[view], OsiINI, TmpCor, EditVersion}, System[view]);
(Only, {}, {}))

Figure 5-8: The View Directory: Top Level

they use is of little interest. Interaction between the *User* views and *System*[*view*] are rigidly restricted, however, as specified by qualifier *Q*16. The qualifiers are explained in Section 5.3.5.

The *User* cluster consists of two sub-clusters, as shown in Fig. 5-9:

- *Public* contains views that are universally available. The units in these views are referred to as *public units*.
- *Restricted* contains views whose use is restricted in one way or another. The units in these views are referred to as *restricted units*.

The arrow specifies that all the *Restricted* views can use all the *Public* views, and *GQ*17 specifies that the public views can use one another.

Fig. 5-10 shows all remaining detail in the *User* cluster.[56] Four public views are differentiated:

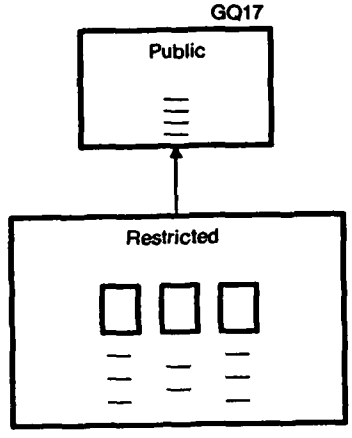
- *Config*[*view*] contains units that describe the system configuration. All units in this view are also in the object cluster *Configuration*.
- *SysInt*[*view*] contains all public units in cluster *Servers* that are system-dependent, but that provide a *system-independent interface* to their users. This view clearly identifies units that would have to be rewritten if Scribe were ported to another system.
- *Variable*[*view*] contains all global variables (units of kind *GlobalVar*).
- *General*[*view*] contains all other public units.

This differentiation is entirely for documentation purposes, but the categories identified are useful, especially *SysInt*.

The role of the restricted views is interesting, and relates to the object structure specified in the object directory. Consider an interaction between two clusters in the object directory, say between *Finished* and *Info* in Fig. 5-5. This interaction specifies that all units within *Finished* can use all units within *Info* that are not hidden. This is an inaccurate picture, however. *Info* maintains information about the current style and environment, which is primarily set up

---

[56] The "[*view*]" suffixes are omitted, to avoid cluttering the figure.



GQ17: ((AlsoHere, Public, Public))

Figure 5-9: The *User* Cluster: Top Level

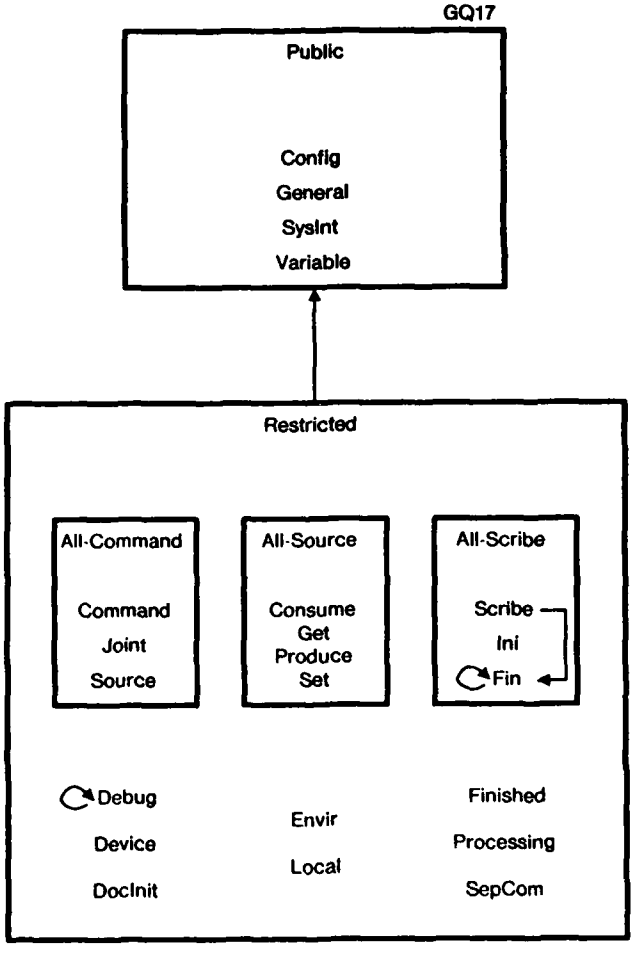


Figure 5-10: The *User* Cluster: Full Detail

by module *command* during the processing of commands; *Finished* merely uses it in the process of producing the finished document. Another way of putting this is that *command* and *Finished* have different views of *Info*: the *command* view consists primarily of units that manipulate the information, whereas the *Finished* view consists primarily of units that provide access to it. The restricted views specify such differences in usage.

A question of granularity arises at this point. It is theoretically possible to associate a view with each module, containing just the units used by that module. A separate view could even be associated with each object, for complete accuracy, giving rise to a vast, square matrix. Such fine granularity tends to lead more to confusion than to clarity. In this particular example, much coarser granularity, involving just large, high-level object clusters, is adequate for expressing all important view restrictions.

Accordingly, certain high-level object clusters are identified as *major*; the major clusters are precisely those shown in Figs. 5-1, 5-2 and 5-3. *Code*, *Core* and *Processing* contain other major clusters; the others do not, and are called *minimal major clusters*. View structure is specified in terms of the major clusters as follows:

- Within each minimal major cluster, arbitrary interactions between *User* views is allowed, including *Restricted* views. These interactions are specified by means of the first group of qualifiers in *GQ*15.
- Across major clusters, *Restricted* views can be used only as explicitly specified in the second group of qualifiers in *GQ*15. The views and their uses are described below.

The *Restricted* views fall into four categories:

- *Local[view]* consists of units used only locally within the minimal major clusters. It includes all hidden units (i.e. units that are not declared to be "global" in Bliss, and hence have *Hidden* qualifiers attached to their nodes in the object directory).
- *Device[view]* consists of units within object cluster *Finished* that deal only with device characteristics and not with the document. Units in

this view cannot be used outside *Finished*, nor can they themselves use restricted units outside *Finished*.

- *All-Source.* These views derive from four possible uses of *Source* that are worth distinguishing. The modules in *Source* read the source input and provide access to parameters that control this process; they also allow text to be inserted into the input, for macro processing and evaluation of string constants, for example, and they allow the control parameters to be set. Hence the following four views:
  - *Consume*[*view*] consists of all units that read input.
  - *Get*[*view*] consists of all units that provide information about control parameters.
  - *Produce*[*view*] consists of all units that insert text into the input.
  - *Set*[*view*] consists of all units that allow the control parameters to be set.

  Different object clusters use different views of *Source*, as specified in *GQ*15.
- Each of the other views is associated with a specific object cluster or set of object clusters; a unit in such a view can be used only by units in the associated object cluster(s) (as well as by other units in its own minimal major cluster):
  - The *All-Command* views consist of units used by modules associated with command execution and related processing: all modules in *Control* and *Gen*, as well as *envir* and *scribe*. Those in *Joint*[*view*] are also used by the object cluster *Finished*, and those in *Source*[*view*] by *Source*, whereas those in *Command*[*view*] are used by neither *Finished* nor *Source*.
  - The *All-Scribe* views consist of units used exclusively[57] by module Scribe, or by each other as indicated by the arrows in the figure. View *Ini*[*view*] contains initialisation routines, *Fin*[*view*] contains finalisation routines, and *Scribe*[*view*] contains a few other units used exclusively by the *scribe* module.
  - *Debug*[*view*] consists of debugging units, used exclusively by object cluster *Debugger* and by each other.
  - *DocInit*[*view*] consists of document initialisation units used exclusively by module *init*.

[57] Throughout this list, "exclusively" refers to interactions across major clusters; interactions within minimal major clusters are always allowed and are therefore not taken into consideration.

- *Envir*[*view*] consists of units used exclusively by module *envir*.
- *Finished*[*view*] consists of units used exclusively by object cluster *Finished*.
- *Processing*[*view*] consists of units within *Control* that are used by units within *Processing*.
- *SepCom*[*view*] consists of units used exclusively by module *sepcom*.

The object/view interactions just described are specified by means of the second group of qualifiers in *GQ*15.

The view directory in this example is especially interesting because it specifies primarily interactions between *objects* and views, rather than between views and views. This phenomenon explains the preponderance of qualifiers: view/view interactions can usually be specified primarily by means of sibling interactions, whereas object/view interactions can only be specified by means of qualifiers. The dominance of object/view interactions arises from the fact that many views are used specifically to identify which units can be used by certain objects, and interactions between those objects and views are then specified directly. An alternative approach is to duplicate units, so that identical units appear in multiple views, and then specify interactions between views instead. This approach was followed in the case of the shared database example, and was explained in Example 2-4. It was not followed in this example for two reasons: to illustrate the alternative approach, and to avoid introducing any units that are not actually part of Scribe.[58]

---

[58] A third approach is perhaps best of all, but is not currently supported by the grid: instead of duplicating units, allowing a single unit to belong to multiple views. The possibility of extending the grid to allow this is discussed further in Section 6.2.

5.3.5. Qualifiers

This section explains all the qualifiers occurring in the grid specification. For convenience, all qualifiers in the object directory are collected together in Fig. 5-11, and all qualifiers in the view directory in Fig. 5-12.

Q1: ⟨(Also, Debugger, Debug[view])⟩

GQ2: ⟨(Also, scribe, All-Scribe)⟩

Q3: ⟨(Also, dvdbl, ZWRString[DVDbl]); (Also, dvxgp, ZWRString[DVXGP])⟩

Q4: ⟨(Only, {msg, osif10, symbol}, scribe)⟩

Q5: ⟨(AlsoHere, msg, driver); (Also, msg, Send); (Only, DecToJulian, DayNum)⟩

Q6: ⟨(Only, {msg, strn}, input)⟩

Q7: ⟨(Also, TextInit, {LINEcreate, HZFcode, ChrWid}); (Also, CMauxfont, DevFont)⟩

Q8: ⟨(Also, DefPart, WHERECMD); (Only, UsePart, Phase)⟩

Q9: ⟨(Also, DefPart, {BoxTest, PageBox, DevCapas, HCURSOR});
(Also, UsePart, {BoxAdd, LINEcreate, PageBox, CVDist, CtrSet});
(Also, PartClose, {DEVmark, Send, QUOTE, XRFMark});
(Only, {}, {})⟩

Q10: ⟨(Also, NOTINI, CtrRegister); (Only, {}, {})⟩

Q11: ⟨(Also, REFevaluate, XRref); (Also, AuxStyle, Send); (Only, {}, {})⟩

Q12: ⟨(AlsoHere, Getevnt, ApEngOrdinal)⟩

Q13: ⟨(Only, gener, {IxActive, IndexClose})⟩

Q14: ⟨(Only, config, StrCon)⟩

Figure 5-11: Qualifiers in the Object Directory

Qualifier *Q*1 specifies that the *Debugger* can use all units in the *Debug*[*view*] of cluster *Code*, even hidden units. Though this is an object/view interaction of the kind generally specified in the view directory, this qualifier is located in the object directory because it also overrides *Hidden* qualifiers in the object

directory.[59] It is an *Also* rather than an *AlsoHere* qualifier, because it affects both directories: it overrides *Hidden* qualifiers in the object directory, and it specifies interactions with a view. Qualifier *GQ2* is similar, specifying that module *scribe* can use all units in the *All-Scribe* views, even hidden units. It is a global qualifier because it applies to all interactions whose sources are units in the *scribe* module, irrespective of their targets.

Qualifier sequence *Q3* specifies two specific interactions between device drivers and hidden units within the *strn* module. These interactions are grave but deliberate violations of information hiding, described further in Section 5.3.6.4.

The qualifier sequences *Q4*, *Q5* and *Q6* restrict the interactions between *Servers* and the rest of Scribe. *Q4* and *Q6* are straightforward; *Q5* reads as follows: "Units in *msg* can use units in *driver* to the extent permitted by the view directory, and units in *msg* can also use *Send*, irrespective of the view directory; otherwise, the only interaction permitted by the object directory is between *DecToJulian* and *DayNum*." These sequences are an interesting illustration of the manner in which restricted interactions are specified. There are, in fact, three variations of the specification "Only those interactions in set *I* are permitted":

1. "All interactions not in *I* are invalid". This variation does *not* state that the interactions in *I* are necessarily valid; their validity depends on other information in both directories. If *I* is sufficiently simple, this restriction can be specified by means of a single *Only* qualifier, as in *Q4* and *Q6*. If it is too complex, a sequence of *AlsoHere*

---

[59] According to the semantics of the grid, it could be located in the view directory nonetheless. I have adopted the convention, however, that qualifiers in the object directory can override the view directory, but not vice versa. This convention is assumed by the prototype implementation, which checks the view directory only in the case of interactions that are found to be valid according to the object directory.

qualifiers terminated by a single *Only* qualifier must be used.[60]

2. "All interactions in *I* are valid according to this directory, and all others are invalid". This variation does state explicitly that the interactions in *I* are valid, but only according to the directory under consideration; the other directory might still find some or all of them to be invalid. This variation is realised by a sequence of one or more *AlsoHere* qualifiers, specifying *I*, terminated by a dummy *Only* qualifier specifying that no other interactions are valid, as in *Q*16.
3. "All interactions in *I* are valid, and all others are invalid". This variation states explicitly that all interactions in *I* are valid, according to both directories; it explicitly overrides any information to the contrary in the other directory. This variation is realised by a sequence of *Also* qualifiers, specifying *I*, followed by a dummy *Only* qualifier, as in *Q*9, *Q*10 and *Q*11.[61]

The variations above can exist in combination in a single qualifier sequence; *Q*5, for example, is a combination of variations 1 and 3. This illustrates yet another interesting point: the use of *AlsoHere* when specifying interactions in terms of clusters belonging to the same directory (*msg*, *driver*), but of *Also* when specifying interactions in terms specific units (*Send*).[62] The rationale behind this is that, whereas the clusters belong to a single directory, a unit belongs to both; an *AlsoHere* qualifier specifying a specific unit will generally have to appear in both directories.

The qualifier sequence *Q*7 specifies additional, extraordinary interactions in

---

[60] If one of the *AlsoHere* qualifiers is applicable, then no further qualifiers are examined, whereas if an interaction is permitted by an *Only* qualifier, further qualifiers *are* examined. Thus a sequence of *AlsoHere* qualifiers followed by *Only* does not precisely achieve the effect of a complex *Only* qualifier; it has some of the flavour of variation 2. This difference is usually not material, but a more complex form of *Only* qualifier to handle all cases of variation 1 would be preferable.

[61] The way in which the three variations are realised in terms of sequences of *Only*, *AlsoHere* and *Also* qualifiers is not ideal. It would be better to have a specific qualifier tailored to each variation, and to allow individual qualifiers to specify arbitrarily complex sets of interactions, thereby removing the need for sequences in this situation. Proposed additions to the collection of qualifiers are discussed in Section 6.1.

[62] There are, of course, also other criteria for choosing between *AlsoHere* and *Also*.

which module *init* engages, due to peculiarities of the document initialisation process. Sequences $Q8$, $Q9$ and $Q10$ specify interactions involving *sepcom*. These are severely restricted and are specified explicitly because they constitute violations of information hiding required by peculiarities of the separate compilation facility of Scribe.

Qualifier sequence $Q11$ specifies the two permitted interactions between *Info* and *Gen*, and $Q13$ specifies the two permitted interactions between *gener* and *Portions*. These interactions are not anomalous; they are merely the only ones permitted. $Q12$ specifies an interaction between a unit within *eccles* and a private unit belonging to *styles*.

$Q14$ specifies that the configuration module *config* can use just the string constructor *StrCon*. This interaction is also not anomalous, though it seems surprising at first sight that the configuration module would use anything at all.

The global qualifier sequence $GQ15$ is the most interesting qualifier sequence in this grid. It specifies most of the view structure of Scribe. The initial *Only* qualifier restricts interactions to *User* views: the *User* views cannot interact with other views except as specified elsewhere ($Q16$). The collection of *AlsoHere* qualifiers following *Only* specify unlimited interaction between *User* views within minimal major clusters. The last one specifies that units in *Gen* also have access to all views within *Info*. The remaining qualifiers specify interactions across major clusters. The *Except* qualifier prevents units in *Device*[*view*] from accessing any restricted units in other major clusters. The collection of *AlsoHere* qualifiers following *Except* specify permitted object/view interactions: the source set of each consists of one or more object clusters, the target set of one or more restricted views.

The question arises of why all the qualifiers in $GQ15$ are in a single, long sequence attached to *User*, rather than being separated and attached to the

```
GQ15: ⟨(Only, User, User);
       (AlsoHere, Debugger, Debugger); (AlsoHere, Control, Control);
       (AlsoHere, Info, Info);         (AlsoHere, Source, Source);
       (AlsoHere, Gen, Gen);           (AlsoHere, Finished, Finished);
       (AlsoHere, sepcom, sepcom);     (AlsoHere, scribe, scribe);
       (AlsoHere, Servers, Servers);   (AlsoHere, Gen, Info);

       (Except, Device[view], Restricted);
       (AlsoHere, {Control, Info, Gen, scribe}, All-Command);
       (AlsoHere, Control, All-Source);
       (AlsoHere, {biblio, styles, strn, Finished}, Consume[view]);
       (AlsoHere, INIT, DocInit);
       (AlsoHere, envir, Envir[view]);
       (AlsoHere, Finished, {Finished[view], Joint[view]});
       (AlsoHere, {styles, msg}, Get[view]);
       (AlsoHere, Processing, Processing[view]);
       (AlsoHere, {envir, Gen, Finished}, Produce[view]);
       (AlsoHere, sepcom, SepCom[view]);
       (AlsoHere, {Gen, msg}, Set[view]);
       (AlsoHere, Source, Source[view])⟩

Q16:  ⟨(AlsoHere, Code, SIX12);
       (AlsoHere, {SysInt[view], OsiINI, TmpCor, EditVersion}, System[view]);
       (Only, {}, {})⟩

GQ17: ⟨(AlsoHere, Public, Public)⟩
```

Figure 5-12: Qualifiers in the View Directory

particular major clusters or views to which they apply. Consider the qualifier (*AlsoHere*, *c*, *v*), specifying an interaction between object cluster *c* and view *v*. It cannot be placed in the object directory, either attached to the node representing *c* or anywhere else, for it would interfere with object/object interactions specified in that directory: it would, for example, override any applicable hidden qualifiers in the object directory.[63] It also cannot be attached to the node in the view directory representing *v*, for it applies to interactions whose *targets* are in *v*, whereas the semantics of the grid are that qualifiers are

[63] A possible solution to this is a new qualifier, *AlsoOther*, which, if attached to a node in the object directory, would override the *view* directory, but not interfere with other information in the object directory. It would thus allow *AlsoHere* qualifiers in the view directory to be placed instead in the object directory when appropriate, and vice versa.

examined only on the paths of *sources*.[64] Attaching all these qualifiers to *User* is therefore the best that can be done in terms of localisation; this does have the advantage that all these qualifiers are in one place, and so can easily be examined together.

*Q*16 specifies all permitted interactions with *System*[*view*], thereby identifying all system dependencies. The first qualifier specifies that any units within *Code* can use *SIX*12, the Bliss debugger. The second qualifier lists the units that are allowed free access to system units. *SysInt*[*view*] contains all public units in *Servers* that are system-dependent. Only three other units can use system units: *OsiINI*, the operating system interface initialisation routine, and *TmpCor* and *EditVersion*, two routines within the *scribe* module.

*Q*17 simply specifies that the *Public* views can use each other freely. This qualifier specifies, in effect, a reflexive sibling interaction involving internal directory nodes; since reflexive sibling interactions are allowed only for leaf nodes, global qualifiers are needed to achieve the same effect at higher levels.

Qualifier sequences *Q*1, *GQ*2, *GQ*15, *GQ*17 and part of *Q*16 specify the view structure of Scribe. The role of each qualifier has been explained in some detail, here and in Section 5.3.4; these explanations could be attached to the qualifiers in the grid.[65] All the other qualifiers specify either restricted interactions or extraordinary interactions. There is a reason for each of these, understood by the author of Scribe; these reasons could also be attached to the qualifiers as documentation. These documented qualifiers would then serve to highlight and explain important structural peculiarities and violations of information hiding.

---

[64] A possible solution to this is to change the semantics so that qualifiers on the paths of targets are considered also. This change has many advantages, and is discussed further in Section 6.2.

[65] The prototype implementation does not currently support the attachment of documentation to grid structures; extending it to do so would be straightforward, and is planned for the near future.

The above discussion of qualifiers revealed some cases where changes in grid semantics or addition of new qualifiers would result in more concise or more elegant specifications. Some suggestions for such changes are made in Chapter 6. Even without them, however, the grid does a good job of specifying the structure of Scribe and identifying structural peculiarities.

5.3.6. Interesting Features

This section draws attention some special structural features of Scribe, and how they are specified by the grid.

5.3.6.1. System Dependencies

The Scribe system makes use of some system routines, and even of machine operations for improved efficiency. However, it does so in a highly disciplined manner, which would make it relatively easy to port Scribe to different systems. The discipline used is not specified in any way in the program, however, except in comments.

The grid specification contains two special views to specify system dependencies explicitly: *System*[*view*] and *SysInt*[*view*]. *System*[*view*] contains all units that form part of the system, and would not exist if Scribe were ported to another machine. *SysInt*[*view*] consists of all units in cluster *Servers* that use *System* units, or that are otherwise system-dependent, such as routines written in assembly language. These would have to be rewritten if the Scribe system were ported. The person charged with rewriting them would not have to search for them; he would merely have to look at the *SysInt*[*view*] row in the matrix.

The fact that *SysInt*[*view*] contains only units in *Servers* highlights the fact that the rest of the Scribe system should be absolutely system-independent. There are in fact two exceptions to this, however:

- The main program itself uses some machine operations. These interactions are specified by means of qualifier sequence $Q16$.

- Units throughout the system call *SIX*12, the Bliss debugger. These interactions are also specified by means of qualifier sequence *Q*16.

Views and qualifiers were thus used to good effect to restrict access to system-dependent facilities, and to specify and document system dependencies that are present.

5.3.6.2. Device Dependencies

Scribe can generate finished output for a variety of devices. Most of the document compilation process is device-independent, however, and Reid was careful to minimise and isolate the code that depends on specific device characteristics. To this end, he wrote a single driver module for each device supported, and a module *dev* to act as an interface between these and the rest of Scribe. The module *dev* exports a collection of abstract device operations, and realises them in terms of actual device operations provided by the device drivers. This discipline cannot be specified in Bliss; it is merely documented by means of comments.

The grid specification uses clusters *Devices* and *DeviceDrivers* to isolate the device drivers, as described in Section 5.3.3.2. Views were thus used to isolate system dependencies, and hidden clusters to isolate device dependencies. Either or both of these methods can generally be used to specify information hiding.

5.3.6.3. The Debugger

In the Scribe source, module *debug* contains a number of routines for displaying data structures belonging to other modules. These routines more properly belong in the modules that own the data structures, but they were placed in *debug* instead so that all aspects of the system to do with debugging would be in one place for easy elimination.

In the grid specification, all such routines were moved to the appropriate modules, but placed in the special view *Debug*[*view*]. This eliminates all

violations of information hiding, while still grouping all debugging routines together for easy elimination. A similar approach was followed in the case of *Ini*[*view*] and *Fin*[*view*], containing initialisation and finalisation routines, though in these cases the primary motivation was documentation.

5.3.6.4. ZWRString

Strings in Scribe are almost always treated as abstract data types maintained by module *strn*, which implements them as null-terminated sequences of characters. There are just two cases in which this implementation is inadequate: two devices, the Diablo and the XGP, require output containing null characters. Rather than introduce a new abstract data type to handle strings with nulls, Reid wrote a routine called *ZWRString* that writes a string maintained by *strn*, but uses a count field to determine how many characters to write instead of stopping at the first null character. This is a violation of the *strn* abstraction, and a dangerous one; so dangerous, in fact, that rather than export *ZWRString* from *strn*, thereby making it publicly available, Reid duplicated it and placed private copies in the device drivers for both the Diablo and the XGP.

Qualifiers in the grid allow *ZWRString* to be treated more elegantly as the special case it is. It is included in module *strn* as a hidden unit,[66] unavailable to users of *strn*, but the device drivers for the Diablo and the XGP are given access to it by means of qualifiers (*Q*3 in Fig. 5-2). The qualifiers provide the desired access, disallow any other accesses, and clearly indicate that an unusual situation exists.

---

[66] Actually, both copies that occur in the code were retained in the grid, so that actual interactions derived from the code could be checked against the grid.

5.3.6.5. Facilities

The notion of associating views with objects or object clusters, as in the case of many of the *Restricted* views, corresponds closely to an approach followed by Reid in the design of Scribe. Once the basic Scribe system existed, Reid added new *facilities* one at a time, such as cross-referencing, separate compilation, etc. Each new facility would have a module of its own to do most of the computation, but would also occasion changes to other modules to provide support for the new facility. The new module corresponds to an object cluster in the grid, the changes in other modules to a view; thus each facility has an associated object cluster and view. The grid specification does indeed have an object cluster associated with each facility, but its granularity is not fine enough to identify all the changes to other modules related to a particular facility, many of which are just a few lines of code within a unit. Nonetheless, the approach is interesting, generally useful, and well suited to specification by means of the grid mechanism.

5.3.6.6. Obsolete Units

The view *Obsolete*[*view*] provides a way of phasing out obsolete units gracefully. Other views can initially be allowed access to it, and it can be allowed access to other views. Units that are to be phased out can then be moved to it without ill-effect. When one is actually ready to eliminate the obsolete units, one can disallow access to *Obsolete*[*view*] by means of a single, trivial change to the grid, and all remaining interactions with obsolete units will be reported as invalid. The version of the Scribe program described still contains a few such interactions.

5.3.7. Checking Interactions

This section describes the use of a slightly modified form of the prototype implementation to check the actual interactions occurring in the Scribe code against the grid specification.

The Scribe source code was processed using a number of Unix shell scripts to isolate information for use by the prototype implementation of the grid:

- All units of the kinds listed in Section 5.3.1. There are 1154 units.
- All actual interactions between units. There are 5032 interactions.

This information was placed in symbolic form in two tables.

The prototype implementation of the grid described in Chapter 4 can process only Modula-2 programs. Minor changes were made to allow it to process the tables of information isolated from Scribe instead.[67] The changes are described here to show how easily the grid implementation can be adapted to a new programming language:

- The nature of *unit descriptors* was changed, to reflect the different kinds of units present in Scribe. This involved changing two modules.
- Changes were made to ignore case in unit identifiers, since case is insignificant in Bliss. This involved changing two modules and introducing two new ones.[68]
- A new tool, called *s2ur*, was written to replace *m2ur*. It creates a unit table and *Uses* relation from the tables of information about the Scribe program.
- Three other modules, affected by the above changes, were recompiled.

---

[67] This involved much less work than having it examine the actual Bliss code, which would have meant incorporating a full Bliss parser into the grid implementation, including symbol table. The objective was to check the interactions in Scribe, not to produce a full grid implementation for Bliss, so the simplest possible approach was followed.

[68] An alternative approach would have been to change four modules and introduce no new ones. That would have lead to the need for far greater recompilation.

The modified implementation was then used to check the actual interactions in the Scribe code against the grid specification described earlier in this chapter. The result was a report of 16 invalid interactions, all deemed by Reid to be obsolete, and hence truly invalid [Reid 84]. The report is shown in Fig. 5.3.7; the token numbers are all zero because source position within file was not determined by the shell scripts used to process the Scribe program.

Checking the 5032 interactions in Scribe took 807 processor seconds on a lightly loaded Vax 11/750. Of this, the core took 377 seconds to perform the actual checks, for a rate of approximately 13 interactions per second.[69] This rate is the same as was measured in the case of the grid specifying the structure of the grid implementation, even though Scribe is a considerably larger program.

5.3.8. Evaluation

Scribe is a large and complex system that has been in widespread use for a number of years. Its author wrote it in a highly disciplined manner, paying careful attention to information hiding and such issues as device-dependence and system-dependence. This discipline is not specified except in comments, and is largely unenforced, because the Bliss language system provides no adequate mechanism for specifying and enforcing it.

The grid mechanism has proven successful at specifying the structure of Scribe. It is able to document and enforce the design discipline. It is able to specify a number of special structural features. The grid is of manageable size and complexity, despite the size of the program; this issue is discussed in detail in section 5.4. Information in the grid is well localised, so that only small chunks need be examined in any particular context; some proposed changes in semantics and qualifiers would improve this situation still further. Even the slow prototype implementation is able to check interactions in the code against the specification with acceptable efficiency.

---

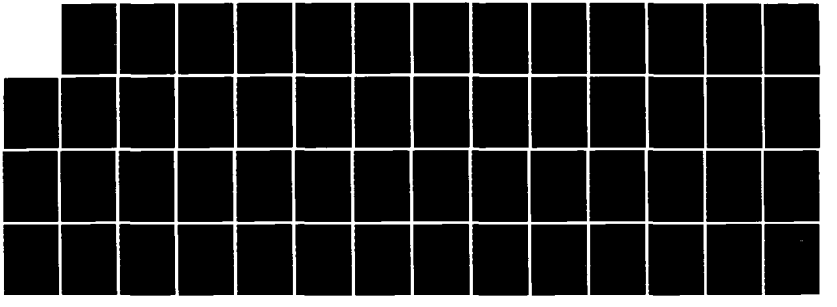
[69] Determined using the stub technique described in Section 5.1.

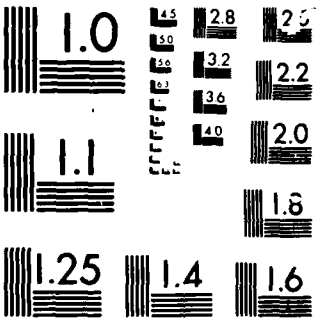
```
Checking Uses

FILE   dyio     TOKEN 0
Invalid interaction: CLOSE Uses RELVECTOR

FILE   ldyio     TOKEN 0
Invalid interaction: CLOSE Uses RELVECTOR

FILE   Command     TOKEN 0
Invalid interaction: CMnewpage Uses PageBrk

FILE   PAGE     TOKEN 0
Invalid interaction: ForcePage Uses PageBrk

FILE   DRIVER     TOKEN 0
Invalid interaction: Main Uses PageBrk

FILE   msgscan     TOKEN 0
Invalid interaction: MSGSCAN Uses EFLOUT

FILE   msgscan     TOKEN 0
Invalid interaction: MSGSCAN Uses FLOUT

FILE   OsiF10     TOKEN 0
Invalid interaction: OsiCldelete Uses STYLVEC

FILE   OsiF10     TOKEN 0
Invalid interaction: OsiTrm Uses DevCapas

FILE   OsiF10     TOKEN 0
Invalid interaction: OsiTrm Uses PaperWidth

FILE   INPUT     TOKEN 0
Invalid interaction: PopFile Uses OUTSWITCH

FILE   INPUT     TOKEN 0
Invalid interaction: PushChannel Uses OUTSWITCH

FILE   INPUT     TOKEN 0
Invalid interaction: PushString Uses OUTSWITCH

FILE   INPUT     TOKEN 0
Invalid interaction: ReadChr Uses OUTSWITCH

FILE   WantFile     TOKEN 0
Invalid interaction: TRYREAD Uses GETVECTOR

FILE   WantFile     TOKEN 0
Invalid interaction: TRYWRITE Uses GETVECTOR


Number of invalid interactions: 16
```

Figure 5-13: Report of Invalid Interactions

AD-A166 937 A NEW PROGRAM STRUCTURING MECHANISM BASED ON LAYERED GRAPHS(U) STANFORD UNIV CA DEPT OF COMPUTER SCIENCE H L OSSHER DEC 84 STAN-CS-85-1070 MDA903-80-C-0432 3/3

UNCLASSIFIED F/G 9/2 NL





MICROCOPY CHART

The exposition of the structure of Scribe in terms of the grid specification shows that the grid provides a suitable framework within which to describe the structure of a large program. The diagrams were derived directly from the grid. The sequence of the exposition was largely dictated by the grid. Each item in the commentary explaining the function of a module or cluster could be attached to the directory node representing that module or cluster. A grid specification with such comments attached would contain all the information in the exposition above, but in a concise, precise and machine-readable form. When examined with a suitable browser, the grid specification should be as readable as the exposition, with the added advantage that the reader would be able to move around as desired, and examine or ignore detail as desired.

Scribe was written long before the grid mechanism was designed. Its author paid careful attention to structure, but did not design Scribe to be either layered or object-oriented, and did not think in terms of multiple views [Reid 84]. Yet even in this case, the dual categorisation of units provided by the grid proved valuable, and the grid successfully satisfied its objectives of specifying, documenting, enforcing and representing program structure. When new programs are written with the aid of the grid, by authors familiar with its features and using the layered, object-oriented programming discipline it was designed to support, it can be expected to perform even better.

5.4. Scaling

Structural information in a grid specification is highly localised: even if the overall size of the specification is large, a reader will need to examine only a small amount of information in any particular context. This is the major factor determining the readability of large grid specifications, and ensures that the specification of even a large program will be manageable. Nonetheless, the overall size of the grid, and the extent to which it grows as program size increases, are of interest. The three examples discussed in this chapter are of

Measure of Size	Database	Grid	Scribe
Lines of program code	—	12000	29000
Units	27	168	1154
Objects	10	67	1154
Actual interactions[70]	45	845	5032
Lines of grid intermediate form[71]	445 (833)	2144 (4403)	5982 (18689)
Directory nodes[71]	10 (20)	50 (117)	104 (1258)
Sibling interactions	31	97	97
Qualifiers[72]	7	46	103

Table 5-5: Sizes of the Examples

Measure of Size	Database	Grid	Scribe
Lines of program code	—	71.4	25.1
Units	1	1	1
Objects	0.4	0.4	1
Actual interactions[70]	1.7	5.0	4.4
Lines of grid intermediate form[71]	16.5 (30.9)	12.8 (26.2)	5.2 (16.2)
Directory nodes[71]	0.4 (0.7)	0.3 (0.7)	0.1 (1.1)
Sibling interactions	1.2	0.6	0.1
Qualifiers[72]	0.2	0.3	0.1

Table 5-6: Per-unit Sizes of the Examples

---

[70] *ImpUses* interactions in the the database example, *Uses* in the others.

[71] Adjusted to remove unit and object overhead, as described in the text. Unadjusted sizes are shown in parentheses.

[72] Excluding *Hidden* qualifiers.

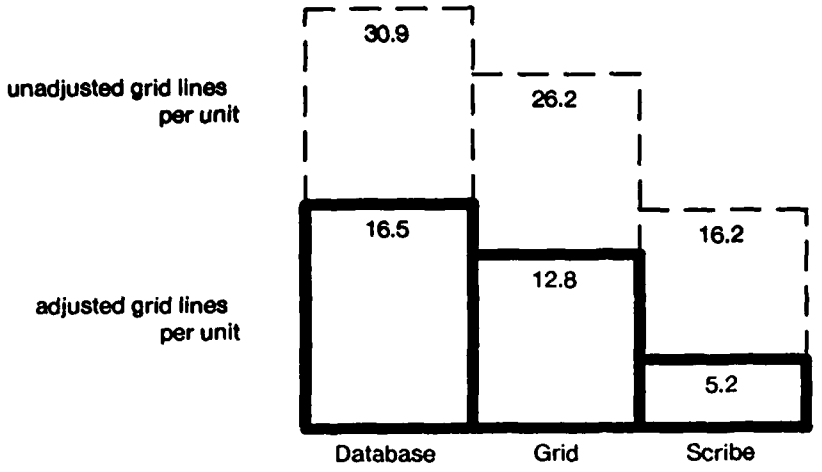


Figure 5-14: Graph of Grid Lines per Unit

widely differing sizes, and hence provide some insight into the growth characteristics of the grid. This section compares the sizes of the example programs and the corresponding grid specifications, and draws conclusions about scaling.

Table 5-5 gives some measures of the sizes of the example programs and their grid specifications. Table 5-6 shows the same measures of size, but normalised by unit. This table therefore gives all sizes relative to the number of units in the program, and can be used to compare grid growth to program growth. Comparison of the figures in Table 5-6 reveals that most measures of per-unit grid size decrease as program size increases, in many cases dramatically. Fig. 5-14 shows this decrease graphically in the case of grid lines per unit. The grid specifications thus grow considerably more slowly than the programs they specify.

Portions of the grid are necessarily linear in program size:

- The matrix contains an entry for every unit.
- The object directory contains a leaf node for every object. Each such node contains several lines of overhead, in addition to any interactions or qualifiers that are present.

The matrix and the object overhead tend to dominate grid size as program size increases, yet contain relatively little structural information: human readers will spend almost all their time examining clusters, sibling interactions and qualifiers. Adjusted size, obtained by removing this overhead, is therefore a superior measure of grid complexity, of greater interest in analysing growth characteristics. Adjusted sizes are shown in the tables, with actual sizes beside them in parentheses.

The numbers of directory nodes (adjusted), sibling interactions and qualifiers are the best measures of grid size, as they are independent of representation. The number of lines in the grid intermediate form gives an exaggerated picture, because that representation has extremely low information density: usually only one item of information appears on a line, and many lines contain just punctuation. It is nonetheless an attractive measure, because it is simple, it encompasses all the others, and it is good for comparison purposes; it was used in Fig. 5-14 for these reasons.

The good growth characteristics of the grid seem to be due primarily to two facts. As the number of units increases:

- The number of interactions per unit tends to remain approximately constant. It tends to depend on the size of individual units rather than on the number of units in the system.
- A single sibling interaction or qualifier can specify an increasing number of interactions. The number of unit-unit interactions specified by a sibling interaction or qualifier depends on the number of units subsidiary to the directory node to which it is attached. This number increases dramatically for high-level nodes as units are added to the program and levels are added to the directories.

Thus, as the number of units increases, the grid becomes more effective at specifying large numbers of interactions concisely, yet it generally does not need to cope with an increasing number of interactions per unit.

The three programs compared above are of widely differing nature, size and

complexity, so they provide a good, though small, sample for the purposes of testing growth characteristics. They indicate that the grid mechanism does indeed scale well, and that total grid size grows slowly relative to program size. This, combined with the localisation of information within a grid specification, makes the grid a viable and attractive structuring mechanism for large programs.

Chapter 6
Directions for Further Research

Experience with the grid mechanism to date has revealed many interesting directions for future research. Some of these are discussed in this chapter. Suggestions on how to proceed, some untested ideas, are included.

6.1. Qualifiers

Qualifiers are intended to handle important special cases in a concise and intuitive manner. As experience is gained with the use of the grid, unanticipated cases and ideas for new or improved qualifiers are likely to arise. The semantic definition of the grid in Section 3.4 is intended to facilitate expansion: it provides a general framework within which many new qualifiers could be inserted with no disruption to existing definitions. Some changes may have to be made to this framework in certain cases, but even that can be done gracefully.

The following is a short list of qualifiers I am currently considering adding to the grid. The precise semantics of these qualifiers have not yet been worked out:

- *OtherDirectory*. The view graph, in specifying interactions between views, effectively specifies the internal structure of each object slice; it assumes that all object slices have essentially the same internal structure, which is true if the layered graph is regular. If the layered graph is not regular, however, the structure of a particular object slice may be markedly different from that specified by the view graph. A good way to handle this case is to associate a different view graph with that object slice, and to use it instead of the standard view graph for all interactions whose sources and/or whose targets are in that object slice. The analogous situation with "object" and "view" exchanged is similar. The most general form of a qualifier to

accomplish this is (*OtherDirectory, d*), which can be placed at any attachment point in the object (view) directory, and specifies that directory *d* is to be used instead of the standard view (object) directory in determining the validity of all interactions to which it applies.

- *Local.* Closely related units are often allowed to use views of each other that are unavailable to outsiders. This situation is covered by the *Same* qualifier in the case where the units all fall in the same object slice or view. The *Local* qualifier is an extension of the *Same* qualifier to the case where they are in the same cluster. Any interactions to which a *Local* qualifier applies are invalid unless the source and target are "sufficiently close" to each other. There are many ways to define "sufficiently close", for example: the source and target must have the same parent, either in both directories, or in the directory containing the qualifier, or in the directory not containing the qualifier; one of their common ancestors must be tagged as defining a "locality"; a number *n* is included in the qualifier, and the paths from both the source and the target to their lowest common ancestor must contain fewer than *n* nodes; etc. A number of variants of this qualifier might be provided, making a variety of options available. At present I tend to favour the tagging scheme, as it is simple, clear and general. If a single tag is too restrictive, multiple tags could be used to convey multiple levels of locality. This meshes well with the qualifiers described next.

- *Tag, SourceTagged, TargetTagged* and *BothTagged.* These qualifiers allow one to specify that certain interactions are valid only if they occur between specified categories of units. Other qualifiers already handle this situation if the categories correspond to clusters; these qualifiers introduce additional classifications that are orthogonal to clusters, and possibly to each other.[73] The qualifier (*Tag, x*) tags a node as belonging to category *x*, but does not directly apply to any interactions. The qualifier (*SourceTagged, x*) specifies that all interactions to which it applies are invalid, except those whose sources are subsidiary to a node tagged *x*. *TargetTagged* and *BothTagged* are similar.

- *AlsoOther.* This is related to *AlsoHere*, but specifies validity in the

---

[73] Because of their ability to specify categories, these qualifiers also specify some organisational details, unlike other qualifiers. It might be better to allow multiple clusterings within a directory instead (i.e. allow directories to be directed acyclic graphs rather than trees), so that clusters will remain the sole means of specifying categories. This issue is discussed further in Section 6.2.

other directory instead of in the current directory. One use is to allow object/view interactions such as those arising in Scribe (Section 5.3.5) to be specified in the object directory without affecting *Hidden* qualifiers.

Of the qualifiers defined in Section 3.3.3, *AlsoHere* and *Also* are termed *positive* because they directly specify interactions that are valid; all the other qualifiers are termed *negative*, because they specify interactions that are *not* valid. Many other positive qualifiers are possible, some of which lead to significant abbreviations. For example:

- *SliceReflexive*. This qualifier specifies that each object slice or view has a sibling interaction with itself. This is a common situation that is currently specified by separate sibling interactions.
- *Reflexive* and *ReflexiveIfTagged*. The qualifier (*Reflexive*) attached to directory node *n* specifies that all interactions whose sources and targets are both subsidiary to *n* are valid. It is thus an abbreviation for the qualifier (*Also*, *n*, *n*). The qualifier (*ReflexiveIfTagged*, *x*) attached to directory node *n* is equivalent to attaching a *Reflexive* qualifier to all nodes tagged *x* that are subsidiary to *n*. This allows a single qualifier to be used to specify that a whole category of clusters, such as those representing modules in the Scribe example (Section 5.3.3.4), are reflexive. "Here" versions, specifying validity only in the current directory, are analogous.
- *Implies*. This qualifier specifies that one kind of interaction implies another (i.e. that one relation is a subset of another). A common case in many languages is that *DefUses* implies *ImpUses*; this qualifier would eliminate the need to specify all *DefUses* interactions separately as *ImpUses* interactions as well.

. The primary difficulty with positive qualifiers such as these is determining what other qualifiers they should override, and when they themselves should be overridden: the first applicable qualifier should override subsequent positive qualifiers in some cases, but apparently not in all (consider the *Implies* qualifier, for example). Positive qualifiers are also responsible for allowing one directory to override another, and great care has to be taken when designing or using them to do this only when appropriate. In short, the semantic framework of the grid as defined in Section 3.4 is geared towards negative qualifiers, and further

research is required to determine how best to integrate positive qualifiers into this framework.

To aid in experimentation with new qualifiers, it would be valuable to define a *qualifier kernel language* that would allow new qualifiers to be constructed from appropriate primitives. Such primitives might include:

- Sets of units and interactions.
- The source and target units and slices.
- The lowest common ancestor, source and target paths, and related concepts.
- The active interaction list and specifier.
- All tags attached to nodes on the source and target paths.
- The current directory and the other directory.
- A *validity pair* specifying validity according to the two directories.
- Control flow primitives, such as conditionals, guards and exits.

For example, the qualifier (*Except*, S_1, S_2) could be defined as something like

if *source-unit* $\in S_1 \wedge$ *target-unit* $\in S_2$ **then**
    validity = (*NotValid*, *NotValid*)
    exit
end if

The interesting research issue is choosing and defining the primitives. Users of the grid should probably not be allowed direct access to these primitives, as they will be powerful and dangerous. Rather, the primitives would be used by a *grid administrator* to define a set of qualifiers, and those qualifiers would be available for general use.

A final issue has to do with the abstract syntax of individual qualifiers. As described in Section 3.3.3, many qualifiers involve pairs of *unit sets*, used to specify sets of interactions. For example, (*Only*, S_1, S_2) specifies that all interactions not in the set

$$I = \{ (u_1, u_2) \mid u_1 \in S_1 \wedge u_2 \in S_2 \}$$

are invalid. Since not all sets of interactions can be specified by means of a single pair of unit sets, sequences of qualifiers might be needed to specify a single restriction. Such sequences are often unattractive and non-intuitive, involving mixtures of *AlsoHere*, *Also* and *Only* qualifiers, and sometimes even dummy qualifiers. This situation was described with reference to the Scribe example in Section 5.3.5. The solution is to allow sequences of pairs of unit sets within single qualifiers. This is an almost trivial extension.

6.2. Semantics

A variety of semantic issues remain to be explored. Those discussed here involve support for approximation, variations and additions to the semantic framework, structural changes to the matrix and directories, and issues to do with the combination of grids.

There is currently no direct support for the technique of approximation in the grid definition: approximation can be achieved only by manually omitting detail from a grid specification. A better approach would be to allow a fully detailed specification to be used in all situations, and to provide a parameter to $valid_{GRID}$ specifying what detail is to be omitted from consideration in the current context. The manner of identifying details to be omitted requires further investigation. One approach is to allow qualifiers and sibling interactions to be named and referred to individually by name; another is to allow them to be tagged according to purpose, level of detail, or other criteria, and be referred to collectively by category. Some combination of these approaches is probably best.

The semantic definition of the grid provides a semantic framework involving four components:

- Active interaction lists.
- Active interaction specifiers.

- Qualifier sequences.
- Exports.

As mentioned in Section 3.4, many variations are possible within this framework. For example, the issue arises of whether always to examine both directories, or to examine one first and then examine the other only if the result returned by the first is *ValidHere*. Always examining both is more general, but less efficient. If one is to be examined first, which should it be?

The adequacy of the framework also needs examination; in particular, the handling of exports could be improved. All the global qualifiers in the active qualifier sequence of an interaction triple are determined by the source unit of the triple alone. This means that interactions can be grouped by source for the purposes of specifying global qualifiers, but not by target, with the result that import-style specifications are heavily favoured. The sole exception is the "Hidden" qualifier, which is handled separately from the active qualifier sequence. A more elegant and general approach would be to have separate active source and active target qualifier sequences; many semantic details remain to be worked out.

Issues to do with the structure of the matrix and directories arise primarily from the desire to share units, nodes or positions in the matrix. There are some situations in which it is convenient to allow the same unit to appear at more than one position in the matrix; the most common example is when several of the views of an object are in fact identical. There are also some situations in which it is convenient to allow multiple units to occupy a single position in the matrix; the most common example is when one wishes to treat definitions and implementations as full-fledged units, and to allow a definition and its implementations to appear together in the matrix. These changes can probably be incorporated into the grid mechanism with few changes to the semantics, though they might have a serious effect on the computational complexity of validity checking.

More serious in its impact is the possibility of allowing a single directory node to appear *in more than one cluster*. This is appealing in some situations, because it allows nodes to be classified in many different ways within a single directory, and because it allows low-level nodes to be shared. The difficulty with this extension is that it violates the tree structure of the directories, on which the semantics of the grid are so heavily based: an interaction triple no longer has a unique lowest common ancestor, sibling ancestor pair or active qualifier sequence. This extension may therefore result in the entire character of the directories being changed.

A less dramatic and perhaps more useful change to the structure of the directories is to allow object slices or views to be associated with internal nodes as well as with leaf nodes. This is particularly useful in cases where a cluster is conceptually associated with a high-level entity, and encapsulates all lower-level entities that are used to implement it. The primary difficulty introduced by this change is that such a cluster, in its capacity as the representative of a slice, can *have sibling interactions with its children*, in addition to the standard sibling interactions it has in its capacity as a cluster. The effects of this need to be investigated, and a suitable means of specifying it needs to be found. It is worth noting that this change can be circumvented by adopting the convention that a leaf node called "self" actually represents its parent, and is associated with the slice that corresponds conceptually to its parent. This convention would be in the abstract syntax only; a concrete syntax could associate slices with internal nodes directly.

The structural issues discussed above are of particular importance in relation to two problems: the handling of definitions and implementations, and the specification of the structure of nested programs. Throughout this thesis, a single definition and all its implementations were considered to be a single unit. The definition subunit was distinguished from the implementation subunits when necessary by the use of two separate relations, *Defl'ses* and *Impl'ses*. This is a

convenient approach in many ways, but I am not convinced it is the best one. One disadvantage is that different implementations of the same definition cannot be distinguished: there is no way to specify that implementation *a* interacts with *x*, but implementation *b* does not. The obvious alternative is to treat each definition and implementation as a separate unit, but then multiple units can occupy a single position in the matrix, which leads to naming and other difficulties.

Many programs whose structure is to be specified do not consist of a flat collection of units, because of nesting or some form of modularisation within the program. A Modula-2 program, for example, is a flat collection of top-level modules, and if one treats top-level modules as atomic, no difficulties arise. This approach was followed in all examples in the thesis, and in the prototype implementation. If one wishes to use the grid mechanism to specify finer-grained structure, however, for example by treating declarations occurring within modules as atomic, one still has to deal with the modules themselves. The obvious approach is to relate modules to clusters, and allow them to be represented by directory nodes. The result is that units are no longer the only components of the grid that correspond directly to entities in the program; the implications of this need to be explored. Even if finer-grained structure is specified in a grid, it may sometimes be convenient to treat modules as atomic. This leads to the notion of an *atomic cluster*. An interesting possibility is to allow certain clusters to be atomic in some views, but not in others.[74]

The issue of combining grids raises many interesting questions. Throughout this thesis, the assumption has been made that a single grid specifies the structure of a whole program or system. (If multiple grids are used to specify the structure of a program, they do so from different points of view; each grid still describes the structure of the whole program.) Since the grid supports top-down

---

[74] I am indebted to Jim Horning for this excellent suggestion.

development and hierarchical decomposition, this is not a serious limitation. However, many situations arise in which a system is not constructed from scratch, but rather is built from existing subsystems. If the structure of the subsystems is already specified by means of grids, it would be desirable to combine these grids to form one describing the entire system. Three general approaches come to mind: nesting, concatenation and merging. Nesting would treat one grid as an atomic unit within another. Concatenation would combine a collection of disjoint grids with different objects and views into a single grid. Merging would use some specified correspondence between slices to merge a collection of related grids into a single grid. The details remain to be worked out, but I suspect that merging is the most promising approach. An interesting related issue is that of using the grid mechanism to specify the structure of program or subroutine libraries.

6.3. Implementation

The prototype implementation of the grid mechanism is rudimentary and inefficient. There are many interesting issues associated with extending it to a full, useful implementation. The following is an incomplete list of possibilities:

- Add a parser for grids, so that users can write grid specifications in a convenient syntax. A graphical user interface would be better still.
- Extend the implementation to handle some of the semantic extensions discussed previously, such as the qualifier kernel language and support for approximation.
- Replace the simple-minded implementations of many of the data structures in the prototype implementation with full, efficient implementations.
- Determine sources of inefficiency and attempt to remove them. This issue is discussed further below.
- Add additional tools, such as a browser, a library manager, a cross-reference generator and a program, along the lines of *make* [Feldman 79], for manipulating programs based on information in the grid.
- Extend the implementation to allow finer granularity by treating individual definitions or declarations as atomic, rather than modules.

- Extend the implementation to handle additional programming languages.

The prototype implementation contains hooks to facilitate all the above extensions. The ultimate goal is to construct a complete, integrated, graphical programming environment based on the grid, including structure-based editors, a version control system, project management aids, documentation aids and the like.

There is much scope for improving the efficiency of the prototype implementation. One serious cause of inefficiency appears to be the reading and writing of the clumsy ASCII intermediate forms using flexible but inefficient high-level input-output routines. Efficiency could be improved somewhat by optimising the routines, and greatly by using binary intermediate forms instead (the ASCII forms should always be available, however, to facilitate debugging and portability). Careful profiling is necessary to determine other bottlenecks. Within the grid core, the techniques of caching and pre-computation should prove useful. It is common to determine the validity of many interactions of one kind before proceeding to consider interactions of another kind; caching could make information about the current kind of interaction readily accessible. Much of the work done by the core involves following paths in the directories in search of global qualifiers; qualifier sequences could be threaded to eliminate such searching, with threads constructed in the background. Many space/time tradeoffs are also possible, and the directories could be optimised for certain common relation kinds, such as *DefUses* and *ImpUses*.

6.4. Object-Oriented Programming with Multiple Views

The primary advantage of the grid mechanism is that it supports a style of programming involving multiple views of objects. There are many situations in which an object is used in different ways by different clients; the examples discussed in this thesis provide several instances. A natural and intuitive way of specifying such multiple usages is by means of multiple views. Identification and separation of views have a number of important advantages:

- *Documentation.* Each view documents a particular usage, showing at a glance how an object appears to a specific client or group of clients.
- *Information hiding.* The view of an object used by a client should contain just those details actually needed by the client; all other details are hidden. This form of information hiding can have vital security implications in situations where multiple clients are allowed to modify a shared structure, but in different ways.
- *Minimisation of recompilation.* Interface changes commonly result in massive recompilation. When multiple views are used, however, one view of an object can often be changed without affecting any other views or their clients, thereby limiting recompilation to units truly affected.

Multiple views of objects also arise from levels of abstraction: each view of an object describes that object at a particular level of abstraction. Splitting a sytem into abstraction levels is a valuable design and implementation technique that is commonly used in many areas, including digital circuit design and communications protocols.

Multiple views of objects are not commonly used in programming today. The chief reason is probably that support for them is inadequate in three areas:

- *Structure specification.* The use of multiple views in a program introduces an extra level of complexity from the point of view of structure specification. In addition to specifying relationships between objects, it is now also necessary to specify relationships between views, and to specify which view of an object is the appropriate one in each context.
- *Programming environments.* Distinct views of an object are often similar, and sometimes even identical. In the absence of assistance from the programming environment, programmers would have to perform a great deal of source-code duplication to create multiple views of objects, and would then have the problem of maintaining multiple similar copies. Programming environment support for these tasks, along the lines of a version control or configuration management system, would significantly reduce these difficulties.
- *Programming languages.* Few programming languages cater directly for the specification of multiple views of a single object, and though the effect can usually be achieved with sufficient "cleverness", the result is often clumsy. Programming language constructs specifically

> designed to allow elegant specification of multiple views are needed. The ability to define one view in terms of another would be particularly valuable.

The grid mechanism was designed to solve the problem of structure specification; the other problems remain to be explored. My expectation is that even simple solutions to these problems will be sufficient to eliminate most duplication and clumsiness, and that programming with multiple views will then become viable and popular.

6.5. Further Generalisation

All work on the grid mechanism to date has dealt with just two partitions of units, always interpreted to mean categorisation by "object" and categorisation by "view". Extension of the grid to handle more than two partitions would be trivial, and might be useful in a number of important situations. For example, introduction of a third directory, called the *version directory*, would allow the grid to specify multiple versions of layered, object-oriented programs. Changing the interpretation of the partitions would require no semantic changes or extensions to the grid at all. It would be interesting to explore other partitions of the units making up computer programs. It would also be interesting to examine the application of the grid to collections of units other than programs, and to fields other than computer science.

Chapter 7
Conclusions

This thesis motivated and described a new program structuring mechanism, called the *grid*. The grid mechanism was designed to meet the following requirements, discussed in detail in Section 1.3:

- *Specification*. It must facilitate the explicit specification of program structure by human beings involved in the development of programs.
- *Documentation*. The specification must be clear and readable, and act as an aid to readers in understanding the program.
- *Enforcement*. The specified structure must be enforced automatically, to ensure both that the specification is accurate and that unintended interactions are not introduced into the program.
- *Representation*. The specified structure must be represented in a form suitable for use by software that is concerned with program structure, such as a browser, editor, compiler or interpreter.
- *Ability to handle layered, object-oriented programs*. The mechanism must be able to handle layered, object-oriented programs well, explicitly identifying the layers and objects and specifying their interactions.
- *Scale*. The mechanism must be able to handle large programs, and structure specifications must not grow unmanageably large and complex as program size increases.

An evaluation of the grid mechanism with respect to these requirements follows.

The grid certainly satisfies the requirements of specification, enforcement and representation. The abstract syntax of the grid, defined in Section 3.3, is a representation of structure suitable for internal use by computers. It presents a uniform interface to any software that needs access to structural information. The semantics of the grid, defined in Section 3.4, ensure that the specified

structure can be enforced. This has been demonstrated by the prototype implementation. Furthermore, the descriptions of the processes of factorisation, clustering and deviation in Chapter 2 showed that a grid can specify arbitrary layered structures, though is is best suited to specifying layered structures that are nearly regular and uniform.

The two-dimensional nature of the grid and the fact that it is based on a layered graph model of program structure make it particularly well suited to handling layered, object-oriented programs. The various examples discussed in this thesis demonstrate its ability to do so. The following factors are of particular importance:

- The matrix identifies the objects and views explicitly, and specifies the object and view associated with each unit.
- The separation of the directories allows one to concentrate on objects independently of views, or views independently of objects.
- The sibling interactions in the object directory specify relationships between objects independently of views, and thus serve to highlight similarities between views. Qualifiers, by specifying exceptions and special cases, serve to highlight differences between views. An analogous statement is true of the view directory.

The requirements that need further consideration are those of documentation and scale. The grid mechanism certainly documents structure, and can handle programs of arbitrary size; the question is whether it is able to document the structure of large programs in a clear and readable manner. Many of its features were designed specifically with such readability in mind:

- Different aspects of structure are specified separately:
    - The entire grid is separate from the program.
    - The dual categorisation of units according to object and view is specified by the matrix.
    - The relationships between objects are specified by the object directory.
    - The relationships between views are specified by the view directory.

- Clustering of objects and views is specified by the hierarchical structure of the directories.
- Unexceptional interactions are specified by sibling interactions.
- Restrictions and exceptional interactions are specified by qualifiers.

Because of this separation, a reader can concentrate on a single aspect without becoming embroiled in the detail of others.

- Structural information is carefully localised, so that only relevant information is encountered in a particular context. The best illustration of this is the fact that a qualifier is attached to the node, interaction list or interaction specifier to which it applies, so that even if a complete grid contains a multitude of qualifiers, only a modest and manageable number will be encountered in any particular context.[75]
- Single qualifiers characterise important restrictions, exceptions and special cases, thereby specifying a great deal of information in a concise, precise and intuitive manner. As experience is gained with the grid and more qualifiers are introduced, the grid is likely to become increasingly effective at handling important structures that occur in practice.
- The hierarchical nature of the directories supports the techniques of hierarchical classification and hierarchical decomposition that are so important to human understanding.
- The techniques of factorisation and clustering allow the grid mechanism to specify arbitrary structures in terms of regular, uniform structures. This has two primary advantages:
  - It identifies structural similarities between different objects and different layers.
  - The regular, uniform structure can be treated as a first approximation of the actual structure. This is valuable because regular, uniform structures are particularly easy to apprehend: relationships between objects are independent of view, relationships between views are independent of object, and all members of a cluster behave in identical fashion relative to other clusters.
- The technique of deviation allows the grid mechanism to specify

---

[75] My intuition is that well-structured programs will actually tend to have a modest total number of qualifiers. This is confirmed by the examples described in Chapter 5.

complex structures in terms of simpler or more familiar structures, with differences clearly identified. This is an especially powerful documentation aid.

- The technique of approximation, by allowing a reader to omit qualifiers from consideration in certain circumstances, further restricts the detail that need be examined in a particular context. In combination with the hierarchical structure of the directories, it allows a reader to proceed from a simple, high-level overview to a detailed, accurate specification by selectively considering more detail as needed.
- The nature of the grid is such that a grid specification can be rendered directly in diagrammatic form; it is, in fact, true to think of the abstract syntax of the grid as a machine-readable encoding of the diagrams.[76] Diagrams are a particularly effective means of documentation for human beings.

The large examples described in Chapter 5 demonstrate that the above features of the grid do indeed lead to readable documentation of large programs. They also provide evidence that the size and complexity of grid specifications grow much more slowly than the size and complexity of the programs they specify, as discussed in Section 5.4. The examples were described in grid terms, using diagrams derived directly from grid specifications; this confirms that the grid provides a suitable framework for describing and explaining the structure of large programs.

Experience to date has shown, therefore, that the grid successfully meets its requirements. In addition, it has the following important properties:

- Multiple grids can be used to specify the structure of a single program from multiple points of view. This is analogous to multiple indexes into a single database.
- Multiple relationships between program units can be specified, either within a single grid, or by means of separate grids.
- The grid can handle multiple, alternative implementations of a single

---

[76] This is the approach I took when designing the grid; I worked solely with diagrams for more than a year before designing the abstract syntax.

specification. Such alternative implementations do not occur frequently, but are useful in some important special cases.

- The grid can be specified as an abstract data type, which then serves as a uniform interface to all software associated with it, such as editors, browsers and compilers.
- The grid is completely programming language independent, and can even be used to specify the structure of entities other than programs. A potential problem with language-independent mechanisms is that the structural information they specify might be inaccessible to the language system (compiler, debugger, etc.). This problem is solved in the case of the grid mechanism by providing an interface through which communication with a language system can take place.
- The syntax of the grid can be tailored to that of any language, allowing it to be integrated gracefully with any existing programming language.
- A grid is textually separate from the program whose structure it specifies. This has several advantages:
    - The structure specification can be perused separately from the program, and be used as an index into the program.
    - Structural information is concentrated in one place, rather than being scattered throughout the program.
- A grid describing program structure can be wholly or partially constructed before the program is written, and then used to control program development.
- A grid can be constructed from an existing program, either completely automatically or with guidance from the user.
- The grid provides a convenient, structured repository for information about a program, such as documentation. This facility can also be used by source management, project management and version control systems, and the like.

These properties enhance the usefulness of the grid, and its suitability as the basis of a comprehensive programming environment.

The usefulness of the grid mechanism in practice depends not only on its effectiveness, but also on the extent to which layered structures are actually used. Layered structures occur whenever levels of abstraction are used, as in the Fable language for which the grid mechanism was designed. Perhaps an even

more fruitful source of layered programs, however, is the style of programming in which each user of an object has its own view of that object. This style of programming is not in widespread use, possibly because it is not viable without sophisticated programming environment support. The grid mechanism is a first step towards providing such support.

The grid mechanism makes two primary contributions to the field of structure specification: it is the first structuring mechanism that can specify, represent, document and enforce layered structures, and it is the first structuring mechanism to make use of the techniques of deviation and approximation. How significant these contributions are remains to be seen.

References

[Ambler, *et al.* 77]
A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch, and R. E. Wells.
GYPSY: A Language for Specification and Implementation of Verifiable Programs.
SIGPLAN Notices 12(3):1-10, March, 1977.
Proceedings of an ACM Conference on Language Design for Reliable Software.

[ANSI 83] American National Standards Institute.
The Programming Language Ada Reference Manual.
Springer-Verlag, 1983.

[Bobrow-Stefik 83]
D. G. Bobrow and M. Stefik.
The LOOPS Manual.
Xerox Palo Alto Research Center, 1983.

[Bobrow-Goldstein 80]
D. G. Bobrow and I. P. Goldstein.
Representing Design Alternatives.
In *An Experimental Description-Based Programming Environment: Four Reports* [Goldstein-Bobrow 81].
1980
Originally published in *Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior*, July, 1980.

[Clarke-Wileden-Wolf 80]
L. A. Clarke, J. C. Wileden, and A. L. Wolf.
Nesting in Ada Programs is for the Birds.
SIGPLAN Notices 15(11):139-145, November, 1980.

[Clarke-Wileden-Wolf 83]
L. A. Clarke, J. C. Wileden, and A. L. Wolf.
Precise Interface Control: System Structure, Language Constructs, and Support Environment.
Technical Report COINS 83-26, University of Massachusetts at Amherst, August, 1983.

[Cooprider 79] L. W. Cooprider.
The Representation of Families of Software Systems.
PhD thesis, Carnegie-Mellon University, 1979.
Published as Technical Report CMU-CS-79-116, Carnegie-Mellon University, April, 1979.

[Cristofor-Wendt-Wonsiewicz 80]
E. Cristofor, T. A. Wendt, and B. C. Wonsiewicz.
Source Control + Tools = Stable Systems.
In *Proceedings of the Fourth Computer Software and Applications Conference*, pages 527-532. IEEE Computer Society, October, 1980.

[Dahl-Myhrhaug-Nygaard 68]
O.-J. Dahl, B. Myhrhaug, and K. Nygaard.
Simula67 Common Base Language.
Technical Report S-2, Norwegian Computing Center, May, 1968.

[DeRemer-Kron 76]
F. DeRemer and H. H. Kron.
Programming-in-the-Large Versus Programming-in-the-Small.
IEEE Transactions on Software Engineering SE-2(2):80-86, June, 1976.

[Deutsch-Taft 80]
L. P. Deutsch and E. A. Taft.
Requirements for an Experimental Programming Environment.
Technical Report CSL-80-10, Xerox Palo Alto Research Center, June, 1980.

[Dickover-McGowan-Ross 77]
M. E. Dickover, C. L. McGowan, and D. T. Ross.
Software Design using SADT.
In *Proceedings of the 1977 Annual Conference*, pages 125-133. ACM, October, 1977.

[Dijkstra 68] E. W. Dijkstra.
The Structure of the T.H.E. Multiprogramming System.
Communications of the ACM 11(5):341-346, May, 1968.

[Dijkstra 72] E. W. Dijkstra.
Notes on Structured Programming.
In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, chapter I. Academic Press, 1972.
1972

[Elliott 84] W. D. Elliott, Xerox Systems Development Division, 3450 Hillview Avenue, Palo Alto, CA 94304.
Personal communication, June, 1984.
June, 1984

[Feldman 79] S. I. Feldman.
Make - A Program for Maintaining Computer Programs.
Software Practice and Experience 9(4):255-265, April, 1979.

[Golden 65] J. T. Golden.
FORTRAN IV Programming and Computing.
Prentice-Hall, 1965.

[Goldstein 81] I. P. Goldstein.
Browsing in a Programming Environment.
In *An Experimental Description-Based Programming Environment: Four Reports* [Goldstein-Bobrow 81].
1981

[Goldstein-Bobrow 80a]
I. P. Goldstein and D. G. Bobrow.
A Layered Approach to Software Design.
Technical Report CSL-80-5, Xerox Palo Alto Research Center, December, 1980.

[Goldstein-Bobrow 80b]
I. P. Goldstein and D. G. Bobrow.
Descriptions for a Programming Environment.
In *An Experimental Description-Based Programming Environment: Four Reports* [Goldstein-Bobrow 81].
1980
Originally published in *Proceedings of the First Annual Conference*, National Association for Artificial Intelligence, August, 1980.

[Goldstein-Bobrow 80c]
I. P. Goldstein and D. G. Bobrow.
Extending Object-Oriented Programming in Smalltalk.
In *An Experimental Description-Based Programming Environment: Four Reports* [Goldstein-Bobrow 81].
1980
Originally published in *Proceedings of the Lisp Conference*, August, 1980.

[Habermann, *et al.* 82]
A. N. Habermann, R. Ellison, R. Medina-Mora, P. Feiler, D. S. Notkin, G. E. Kaiser, D. B. Garlan, and S. Popovich.
The Second Compendium of Gandalf Documentation.
Technical Report, Carnegie-Mellon University, May, 1982.

[Hoare 68] C. A. R. Hoare.
Record Handling, page 292.
In F. Genuys (editor), *Programming Languages*, pages 291-347. Academic Press, 1968.

[Horning-Randell 73]
J. J. Horning and B. Randell.
Process Structuring.
Computing Surveys 5(1):5-30, March, 1973.

[Ingalls 78] D. H. Ingalls.
The Smalltalk-76 Programming System: Design and Implementation.
In *Conference Record of the Fifth Annual Symposium on Principles of Programming Languages*, pages 9-16. ACM, January, 1978.

[Jensen-Wirth 78]
K. Jensen and N. Wirth.
Pascal User Manual and Report.
Second edition, Springer-Verlag, 1978.

[Kaiser-Habermann 82]
G. E. Kaiser and A. N. Habermann.
An Environment for System Version Control.
In *The Second Compendium of Gandalf Documentation* [Habermann, *et al.* 82].
1982

[Kernighan-Mashey 81]
B. W. Kernighan and J. R. Mashey.
The Unix Programming Environment.
Computer 14(4):12-24, April, 1981.

[Knuth 73] D. E. Knuth.
The Art of Computer Programming. Volume 1: *Fundamental Algorithms.*
Addison-Wesley, 1973.

[Lampson-Schmidt 83]
B. W. Lampson and E. E. Schmidt.
Practical Use of a Polymorphic Applicative Language.
In *Conference Record of the Tenth Annual Symposium on Principles of Programming Languages*, pages 237-255.
ACM, January, 1983.

[Linton 83] M. A. Linton.
Queries and Views of Programs Using a Relational Database System.
PhD thesis, University of California at Berkeley, 1983.
Published as Technical Report UCB/CSD 83/164, University of California at Berkeley, December, 1983.

[Liskov, *et al.* 77]
B. H. Liskov, A. Snyder, R. Atkinson, and J. C. Schaffert.
Abstraction Mechanisms in CLU.
Communications of the ACM 20(8):564-576, August, 1977.

[Liskov, *et al.* 81]
B. H. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder.
CLU Reference Manual.
Springer-Verlag, 1981.

[Mitchell-Maybury-Sweet 79]
J. G. Mitchell, W. Maybury, and R. E. Sweet.
Mesa Language Manual.
Technical Report CSL-79-3, Xerox Palo Alto Research Center, April, 1979.

[Naur 63] P. Naur (editor).
Revised Report on the Algorithmic Language ALGOL60.
Communications of the ACM 6(1):1-17, 1963.

[Nestor-Wulf-Lamb 81]
J. R. Nestor, W. A. Wulf, and D. A. Lamb.
IDL - Interface Description Language: Formal Description.
Technical Report CMU-CS-81-139, Carnegie-Mellon University, August, 1981.

[Ossher-Reid 83] H. L. Ossher and B. K. Reid.
Fable: A Programming Language Solution to IC Process Automation Problems.
SIGPLAN Notices 18(6):137-148, June, 1983.
Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems.

[Parnas 71] D. L. Parnas.
Information Distribution Aspects of Design Methodology.
In C. R. Freeman (editor), *Information Processing 71: Proceedings of IFIP Congress 71.* Volume 1: *Foundations and Systems*, pages 339-344. IFIP, August, 1971.

[Parnas 74] D. L. Parnas.
On a 'Buzzword': Hierarchical Structure.
In J. L. Rosenfeld (editor), *Information Processing 74: Proceedings of IFIP Congress 74*, pages 336-339. IFIP, August, 1974.

[Project MAC 74]
Project MAC.
Introduction to MULTICS.
Technical Report MAC TR-123, Massachusetts Institute of Technology, February, 1974.

[Reid 78] B. K. Reid.
Scribe User Manual.
Carnegie-Mellon University, 1978.
This is an outdated edition of the manual, but it corresponds to the version of the system for which the grid specification was written.

[Reid 80] B. K. Reid.
Scribe: A Document Specification Language and its Compiler.
PhD thesis, Carnegie-Mellon University, 1980.
Published as Technical Report CMU-CS-81-100, Carnegie-Mellon University, October, 1980.

[Reid 84] B. K. Reid, Computer Systems Laboratory, Stanford University.
Personal communication, August-September, 1984.
1984

[Reid-Walker 80] B. K. Reid and J. H. Walker.
Scribe User Manual.
Third edition, Unilogic, 1980.

[Ritchie-Thompson 74]
D. M. Ritchie and K. Thompson.
The UNIX Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.

[Ross 77] D. T. Ross.
Structured Analysis (SA): A Language for Communicating Ideas.
IEEE Transactions on Software Engineering SE-3(1):16-34, January, 1977.

[Ross-Schoman 77]
D. T. Ross and K. E. Schoman, Jr.
Structured Analysis for Requirements Definition.
IEEE Transactions on Software Engineering SE-3(1):6-15, January, 1977.

[Schmidt 82] E. E. Schmidt.
Controlling Large Software Development in a Distributed Environment.
Technical Report CSL-82-7, Xerox Palo Alto Research Center, December, 1982.

[Teitelman 78] W. Teitelman.
Interlisp Reference Manual.
Xerox Palo Alto Research Center, 1978.

[Teitelman-Masinter 81]
W. Teitelman and L. Masinter.
The Interlisp Programming Environment.
Computer 14(4):25-33, April, 1981.

[Tichy 80] W. F. Tichy.
Software Development Control Based on System Structure Description.
PhD thesis, Carnegie-Mellon University, 1980.
Published as Technical Report CMU-CS-80-120, Carnegie-Mellon University, January, 1980.

[Ullman 82] J. D. Ullman.
Principles of Database Systems.
Second edition, Computer Science Press, 1982.

[Wirth 83] N. Wirth.
Programming in Modula-2.
Second edition, Springer-Verlag, 1983.

[Woitok 83] R. Woitok.
Abstracting Linked Data Structures Using Incremental Records.
SIGPLAN Notices 18(11):54-63, November, 1983.

[Wolf-Clarke-Wileden 83]
A. L. Wolf, L. A. Clarke, and J. C. Wileden.
A Formalism for Describing and Evaluating Visibility Control Mechanisms.
Technical Report COINS 83-34, University of Massachusetts at Amherst, October, 1983.

[Wulf-London-Shaw 76]
W. A. Wulf, R. L. London, and M. Shaw.
An Introduction to the Construction and Verification of ALPHARD Programs.
IEEE Transactions on Software Engineering SE-2(4):253-265, December, 1976.

[Wulf-Russell-Habermann 71]
W. A. Wulf, D. B. Russell, and A. N. Habermann.
BLISS: A Language for Systems Programming.
Communications of the ACM 14(12):780-790, December, 1971.

Appendix A
The Shared Database Example

This appendix contains a selection of computer input used and output produced by the prototype implementation when processing the shared database example. Representative extracts of the intermediate forms of all structures are included; the full text is available from me on request. The appendix contains the following material:

A.1. The Program

The skeleton Modula-2 program for the shared database example is:

```
DEFINITION MODULE A;
END A.

IMPLEMENTATION MODULE A;
    IMPORT DBA;
    IMPORT VA;
    IMPORT ALA;
    IMPORT LA;
    IMPORT PA;
    IMPORT SA;
END A.

DEFINITION MODULE B;
END B.

IMPLEMENTATION MODULE B;
    IMPORT DBB;
    IMPORT VB;
    IMPORT ALB;
    IMPORT LB;
    IMPORT PB;
    IMPORT SB;
END B.

DEFINITION MODULE DB;
    IMPORT VDB;
    IMPORT SDB;
END DB.

IMPLEMENTATION MODULE DB;
    IMPORT H;
    IMPORT VDB;
    IMPORT ALDB;
    IMPORT SDB;
END DB.

DEFINITION MODULE DBA;
    IMPORT VA;
    IMPORT SA;
END DBA.

IMPLEMENTATION MODULE DBA;
    IMPORT DB;
    IMPORT VA;
    IMPORT SA;
END DBA.

DEFINITION MODULE DBB;
    IMPORT VB;
    IMPORT SB;
END DBB.

IMPLEMENTATION MODULE DBB;
    IMPORT DB;
    IMPORT VB;
    IMPORT SB;
END DBB.

DEFINITION MODULE H;
END H.

IMPLEMENTATION MODULE H;
    IMPORT R;
END H.

DEFINITION MODULE R;
END R.

IMPLEMENTATION MODULE R;
END R.

DEFINITION MODULE V;
    IMPORT AL;
    IMPORT L;
    IMPORT P;
    IMPORT S;
END V.

IMPLEMENTATION MODULE V;
    IMPORT AL;
    IMPORT L;
    IMPORT P;
    IMPORT S;
END V.

DEFINITION MODULE VA;
    IMPORT ALA;
    IMPORT LA;
    IMPORT PA;
    IMPORT SA;
END VA.

IMPLEMENTATION MODULE VA;
    IMPORT V;
END VA.

DEFINITION MODULE VB;
    IMPORT ALB;
    IMPORT LB;
    IMPORT PB;
    IMPORT SB;
END VB.

IMPLEMENTATION MODULE VB;
    IMPORT V;
END VB.
```

```
DEFINITION MODULE VDB;
    IMPORT ALDB;
    IMPORT LDB;
    IMPORT PDB;
    IMPORT SDB;
END VDB.

IMPLEMENTATION MODULE VDB;
    IMPORT V;
END VDB.

DEFINITION MODULE AL;
END AL.

IMPLEMENTATION MODULE AL;
    IMPORT L;
    IMPORT P;
END AL.

DEFINITION MODULE ALA;
END ALA.

IMPLEMENTATION MODULE ALA;
    IMPORT AL;
END ALA.

DEFINITION MODULE ALB;
END ALB.

IMPLEMENTATION MODULE ALB;
    IMPORT AL;
END ALB.

DEFINITION MODULE ALDB;
END ALDB.

IMPLEMENTATION MODULE ALDB;
    IMPORT AL;
END ALDB.

DEFINITION MODULE L;
END L.

IMPLEMENTATION MODULE L;
    IMPORT P;
END L.

DEFINITION MODULE LA;
END LA.

IMPLEMENTATION MODULE LA;
    IMPORT L;
END LA.

DEFINITION MODULE LB;
END LB.

IMPLEMENTATION MODULE LB;
    IMPORT L;
END LB.

DEFINITION MODULE LDB;
END LDB.

IMPLEMENTATION MODULE LDB;
    IMPORT L;
END LDB.

DEFINITION MODULE P;
END P.

IMPLEMENTATION MODULE P;
END P.

DEFINITION MODULE PA;
END PA.

IMPLEMENTATION MODULE PA;
    IMPORT P;
END PA.

DEFINITION MODULE PB;
END PB.

IMPLEMENTATION MODULE PB;
    IMPORT P;
END PB.

DEFINITION MODULE PDB;
END PDB.

IMPLEMENTATION MODULE PDB;
    IMPORT P;
END PDB.

DEFINITION MODULE S;
END S.

IMPLEMENTATION MODULE S;
END S.

DEFINITION MODULE SA;
END SA.

IMPLEMENTATION MODULE SA;
    IMPORT S;
END SA.

DEFINITION MODULE SB;
END SB.

IMPLEMENTATION MODULE SB;
    IMPORT S;
END SB.

DEFINITION MODULE SDB;
END SDB.

IMPLEMENTATION MODULE SDB;
    IMPORT S;
END SDB.
```

A.2. The Relation "DefUses" in Intermediate Form

The intermediate form of the relation *DefUses*, produced from the program by *m2ur*, is:

```
Versions [
        Identifier      0
        Relation        2
        UnitTable       1
        UnitName        0
        ProgramInfo     0
        SourcePosition  1
        ]

Relation [
        id              DefUses;
        uTab            UnitTableRef [
                id              Units;
                directory       utab
                ];
        interactions    {
                interaction [
                        source          6;
                        destination     11;
                        position        SourcePosition [
                                fileName        ./DB.def;
                                tokenNumber     6
                                ]
                        ]
                interaction [
                        source          6;
                        destination     27;
                        position        SourcePosition [
                                fileName        ./DB.def;
                                tokenNumber     9
                                ]
                        ]
                interaction [
                        source          4;
                        destination     9;
                        position        SourcePosition [
                                fileName        ./DBA.def;
                                tokenNumber     6
                                ]
                        ]
                interaction [
                        source          4;
                        destination     25;
                        position        SourcePosition [
```

```
                        fileName        ./DBA.def;
                        tokenNumber     9
                        ]
                ]
        interaction [
                source          5;
                destination     10;
                position        SourcePosition [
                        fileName        ./DBB.def;
                        tokenNumber     6
                        ]
                ]
        interaction [
                source          5;
                destination     26;
                position        SourcePosition [
                        fileName        ./DBB.def;
                        tokenNumber     9
                        ]
                ]

. . .  104 lines omitted . . .

        interaction [
                source          11;
                destination     19;
                position        SourcePosition [
                        fileName        ./VDB.def;
                        tokenNumber     9
                        ]
                ]
        interaction [
                source          11;
                destination     23;
                position        SourcePosition [
                        fileName        ./VDB.def;
                        tokenNumber     12
                        ]
                ]
        interaction [
                source          11;
                destination     27;
                position        SourcePosition [
                        fileName        ./VDB.def;
                        tokenNumber     15
                        ]
                ]
        }
]
```

A.3. The Relation "ImpUses" in Intermediate Form

The intermediate form of the relation *ImpUses*, produced from the program by *m2ur*, is:

```
Versions [
        Identifier      0
        Relation        2
        UnitTable       1
        UnitName        0
        ProgramInfo     0
        SourcePosition  1
        ]

Relation [
        id              ImpUses;
        uTab            UnitTableRef [
                id              Units;
                directory       utab
                ];
        interactions    {
                interaction [
                        source          2;
                        destination     4;
                        position        SourcePosition [
                                fileName        ./A.mod;
                                tokenNumber     6
                                ]
                        ]
                interaction [
                        source          2;
                        destination     9;
                        position        SourcePosition [
                                fileName        ./A.mod;
                                tokenNumber     9
                                ]
                        ]
                interaction [
                        source          2;
                        destination     13;
                        position        SourcePosition [
                                fileName        ./A.mod;
                                tokenNumber     12
                                ]
                        ]
                interaction [
                        source          2;
                        destination     17;
                        position        SourcePosition [
```

```
                        fileName        ./A.mod;
                        tokenNumber     15
                        ]
                ]
        interaction [
                source          2;
                destination     21;
                position        SourcePosition [
                        fileName        ./A.mod;
                        tokenNumber     18
                        ]
                ]
        interaction [
                source          2;
                destination     25;
                position        SourcePosition [
                        fileName        ./A.mod;
                        tokenNumber     21
                        ]
                ]

. . . 296 lines omitted . . .

        interaction [
                source          10;
                destination     12;
                position        SourcePosition [
                        fileName        ./VB.mod;
                        tokenNumber     6
                        ]
                ]
        interaction [
                source          11;
                destination     12;
                position        SourcePosition [
                        fileName        ./VDB.mod;
                        tokenNumber     6
                        ]
                ]
        }
]
```

A.4. Listings of Relations

The listing of *DefUses* produced by *rlist* is shown below. The numbers in parentheses are the atom names of the units, listed as a debugging aid.

```
Listing DefUses

(DB       -> VDB     ) = (6 -> 11)      FILE   ./DB.def     TOKEN 6
(DB       -> SDB     ) = (6 -> 27)      FILE   ./DB.def     TOKEN 9
(DBA      -> VA      ) = (4 -> 9)       FILE   ./DBA.def     TOKEN 6
(DBA      -> SA      ) = (4 -> 25)      FILE   ./DBA.def     TOKEN 9
(DBB      -> VB      ) = (5 -> 10)      FILE   ./DBB.def     TOKEN 6
(DBB      -> SB      ) = (5 -> 26)      FILE   ./DBB.def     TOKEN 9
(V        -> AL      ) = (12 -> 16)     FILE   ./V.def     TOKEN 6
(V        -> L       ) = (12 -> 20)     FILE   ./V.def     TOKEN 9
(V        -> P       ) = (12 -> 24)     FILE   ./V.def     TOKEN 12
(V        -> S       ) = (12 -> 28)     FILE   ./V.def     TOKEN 15
(VA       -> ALA     ) = (9 -> 13)      FILE   ./VA.def     TOKEN 6
(VA       -> LA      ) = (9 -> 17)      FILE   ./VA.def     TOKEN 9
(VA       -> PA      ) = (9 -> 21)      FILE   ./VA.def     TOKEN 12
(VA       -> SA      ) = (9 -> 25)      FILE   ./VA.def     TOKEN 15
(VB       -> ALB     ) = (10 -> 14)     FILE   ./VB.def     TOKEN 6
(VB       -> LB      ) = (10 -> 18)     FILE   ./VB.def     TOKEN 9
(VB       -> PB      ) = (10 -> 22)     FILE   ./VB.def     TOKEN 12
(VB       -> SB      ) = (10 -> 26)     FILE   ./VB.def     TOKEN 15
(VDB      -> ALDB    ) = (11 -> 15)     FILE   ./VDB.def     TOKEN 6
(VDB      -> LDB     ) = (11 -> 19)     FILE   ./VDB.def     TOKEN 9
(VDB      -> PDB     ) = (11 -> 23)     FILE   ./VDB.def     TOKEN 12
(VDB      -> SDB     ) = (11 -> 27)     FILE   ./VDB.def     TOKEN 15
```

The listing of *DefUses* produced by *rlist*1 is:

```
DBA:        VA SA

DBB:        VB SB

DB:         VDB SDB

VA:         ALA LA PA SA

VB:         ALB LB PB SB

VDB:        ALDB LDB PDB SDB

V:          AL L P S
```

The inverted listing of *DefUses* produced by *rlist2* is:

```
VA:          DBA
VB:          DBB
VDB:         DB
ALA:         VA
ALB:         VB
ALDB:        VDB
AL:          V
LA:          VA
LB:          VB
LDB:         VDB
L:           V
PA:          VA
PB:          VB
PDB:         VDB
P:           V
SA:          DBA VA
SB:          DBB VB
SDB:         DB VDB
S:           V
```

The listing of *ImpUses* produced by *rlist* is:

```
Listing ImpUses

(A        -> DBA     ) = (2 -> 4)       FILE   ./A.mod     TOKEN 6
(A        -> VA      ) = (2 -> 9)       FILE   ./A.mod     TOKEN 9
(A        -> ALA     ) = (2 -> 13)      FILE   ./A.mod     TOKEN 12
(A        -> LA      ) = (2 -> 17)      FILE   ./A.mod     TOKEN 15
(A        -> PA      ) = (2 -> 21)      FILE   ./A.mod     TOKEN 18
(A        -> SA      ) = (2 -> 25)      FILE   ./A.mod     TOKEN 21
(AL       -> L       ) = (16 -> 20)     FILE   ./AL.mod     TOKEN 6
(AL       -> P       ) = (16 -> 24)     FILE   ./AL.mod     TOKEN 9
(ALA      -> AL      ) = (13 -> 16)     FILE   ./ALA.mod     TOKEN 6
(ALB      -> AL      ) = (14 -> 16)     FILE   ./ALB.mod     TOKEN 6
(ALDB     -> AL      ) = (15 -> 16)     FILE   ./ALDB.mod     TOKEN 6
(B        -> DBB     ) = (3 -> 5)       FILE   ./B.mod     TOKEN 6
(B        -> VB      ) = (3 -> 10)      FILE   ./B.mod     TOKEN 9
(B        -> ALB     ) = (3 -> 14)      FILE   ./B.mod     TOKEN 12
(B        -> LB      ) = (3 -> 18)      FILE   ./B.mod     TOKEN 15
(B        -> PB      ) = (3 -> 22)      FILE   ./B.mod     TOKEN 18
(B        -> SB      ) = (3 -> 26)      FILE   ./B.mod     TOKEN 21
(DB       -> H       ) = (6 -> 7)       FILE   ./DB.mod     TOKEN 6
(DB       -> VDB     ) = (6 -> 11)      FILE   ./DB.mod     TOKEN 9
(DB       -> ALDB    ) = (6 -> 15)      FILE   ./DB.mod     TOKEN 12
(DB       -> SDB     ) = (6 -> 27)      FILE   ./DB.mod     TOKEN 15
(DBA      -> DB      ) = (4 -> 6)       FILE   ./DBA.mod     TOKEN 6
(DBA      -> VA      ) = (4 -> 9)       FILE   ./DBA.mod     TOKEN 9
(DBA      -> SA      ) = (4 -> 25)      FILE   ./DBA.mod     TOKEN 12
(DBB      -> DB      ) = (5 -> 6)       FILE   ./DBB.mod     TOKEN 6
(DBB      -> VB      ) = (5 -> 10)      FILE   ./DBB.mod     TOKEN 9
(DBB      -> SB      ) = (5 -> 26)      FILE   ./DBB.mod     TOKEN 12
(H        -> R       ) = (7 -> 8)       FILE   ./H.mod     TOKEN 6
(L        -> P       ) = (20 -> 24)     FILE   ./L.mod     TOKEN 6
(LA       -> L       ) = (17 -> 20)     FILE   ./LA.mod     TOKEN 6
(LB       -> L       ) = (18 -> 20)     FILE   ./LB.mod     TOKEN 6
(LDB      -> L       ) = (19 -> 20)     FILE   ./LDB.mod     TOKEN 6
(PA       -> P       ) = (21 -> 24)     FILE   ./PA.mod     TOKEN 6
(PB       -> P       ) = (22 -> 24)     FILE   ./PB.mod     TOKEN 6
(PDB      -> P       ) = (23 -> 24)     FILE   ./PDB.mod     TOKEN 6
(SA       -> S       ) = (25 -> 28)     FILE   ./SA.mod     TOKEN 6
(SB       -> S       ) = (26 -> 28)     FILE   ./SB.mod     TOKEN 6
(SDB      -> S       ) = (27 -> 28)     FILE   ./SDB.mod     TOKEN 6
(V        -> AL      ) = (12 -> 16)     FILE   ./V.mod     TOKEN 6
(V        -> L       ) = (12 -> 20)     FILE   ./V.mod     TOKEN 9
(V        -> P       ) = (12 -> 24)     FILE   ./V.mod     TOKEN 12
(V        -> S       ) = (12 -> 28)     FILE   ./V.mod     TOKEN 15
(VA       -> V       ) = (9 -> 12)      FILE   ./VA.mod     TOKEN 6
(VB       -> V       ) = (10 -> 12)     FILE   ./VB.mod     TOKEN 6
(VDB      -> V       ) = (11 -> 12)     FILE   ./VDB.mod     TOKEN 6
```

The listing of *ImpUses* produced by *rlist1* is:

```
A:          DBA VA ALA LA PA SA
B:          DBB VB ALB LB PB SB
DBA:        DB VA SA
DBB:        DB VB SB
DB:         H VDB ALDB SDB
H:          R
VA:         V
VB:         V
VDB:        V
V:          AL L P S
ALA:        AL
ALB:        AL
ALDB:       AL
AL:         L P
LA:         L
LB:         L
LDB:        L
L:          P
PA:         P
PB:         P
PDB:        P
SA:         S
SB:         S
SDB:        S
```

The inverted listing of *ImpUses* produced by *rlist2* is:

```
DBA:        A
DBB:        B
DB:         DBA DBB
H:          DB
R:          H
VA:         A DBA
VB:         B DBB
VDB:        DB
V:          VA VB VDB
ALA:        A
ALB:        B
ALDB:       DB
AL:         V ALA ALB ALDB
LA:         A
LB:         B
L:          V AL LA LB LDB
PA:         A
PB:         B
P:          V AL L PA PB PDB
SA:         A DBA
SB:         B DBB
SDB:        DB
S:          V SA SB SDB
```

A.5. The Unit Table in Intermediate Form

The intermediate form of the unit table corresponding to the shared database program, produced from the program by *m2ur*, is shown below. An old version of the intermediate form, in which some fields are not identified by name, is still used for unit tables. The number "28" appearing just after the version information is the last atom name used; atom names are just small positive integers. Each line of four unnamed fields, together with the "ProgramInfo" after it, corresponds to a single unit. The first field is the unit identifier, the second is the atom name, and the other two are links used within the table. The unit "top" is a dummy unit that serves as the root of the table.

```
Versions [
        Identifier      0
        Relation        2
        UnitTable       1
        UnitName        0
        ProgramInfo     0
        SourcePosition  1
        ]

28
top     1       28      0
        ProgramInfo [
                kind    Unknown
                ]
A       2       0       0
        ProgramInfo [
                kind    ModulePair;
                defPos  SourcePosition [
                        fileName        ./A.def;
                        tokenNumber     0
                        ];
                impPos  SourcePosition [
                        fileName        ./A.mod;
                        tokenNumber     0
                        ]
                ]
B       3       0       2
        ProgramInfo [
                kind    ModulePair;
                defPos  SourcePosition [
                        fileName        ./B.def;
                        tokenNumber     0
```

```
                    ];
                impPos  SourcePosition [
                        fileName        ./B.mod;
                        tokenNumber     0
                        ]
                ]
DBA     4       0       3
        ProgramInfo [
                kind    ModulePair;
                defPos  SourcePosition [
                        fileName        ./DBA.def;
                        tokenNumber     0
                        ];
                impPos  SourcePosition [
                        fileName        ./DBA.mod;
                        tokenNumber     0
                        ]
                ]
DBB     5       0       4
        ProgramInfo [
                kind    ModulePair;
                defPos  SourcePosition [
                        fileName        ./DBB.def;
                        tokenNumber     0
                        ];
                impPos  SourcePosition [
                        fileName        ./DBB.mod;
                        tokenNumber     0
                        ]
                ]

. . . 264 lines omitted . . .

S       28      0       27
        ProgramInfo [
                kind    ModulePair;
                defPos  SourcePosition [
                        fileName        ./S.def;
                        tokenNumber     0
                        ];
                impPos  SourcePosition [
                        fileName        ./S.mod;
                        tokenNumber     0
                        ]
                ]
```

A.6. The Grid in Intermediate Form

The intermediate form of the grid is shown below. This form was constructed by hand using a text editor.

```
Versions [
    Node        1
    ]

Grid [
    NodeTab             NodeTable [
            lastNode    20;
            entries             <

Node [
    Name                1;
    Id          groups;[77]
    Parent              0;
    FirstChild  2;
    NextSibling 0;
    Vector[78]      0;
    Exported    TRUE;
    Interactions        <
        >;
    OtherDirectory      Directory [
            Root                16;
            LeafTab             LeafTable [
                    entries <
                            1 : 17
                            2 : 18
                            3 : 19
                            4 : 20
                            >
                    ]
            ];
    GeneralQualifiers[79]     <
        Qualifier [
            Kind                NonUnitReflexive
            ]
        Qualifier [
```

---

[77] "Group" is an obsolete term for "object slice".

[78] "Vector" is an obsolete term for "slice".

[79] "General qualifier" is an obsolete term for "global qualifier".

```
            Kind                Same
            ]
        >
    ]

Node [
    Name                2;
    Id          users;
    Parent              1;
    FirstChild  4;
    NextSibling 3;
    Vector              0;
    Exported    TRUE;
    Interactions        <
            InteractionList [
                    RelationId  ImpUses;
                    Entries             <
                            Interaction [
                                    Sibling     3
                                    ]
                            >
                    ]
            >
    ]


. . .  46 lines omitted . . .

Node [
    Name                6;
    Id          db;
    Parent              3;
    FirstChild  0;
    NextSibling 7;
    Vector              3;
    Exported    TRUE;
    Interactions        <
            InteractionList [
                    RelationId  DefUses;
                    Entries             <
                            Interaction [
                                    Sibling     8
                                    ]
                            Interaction [
                                    Sibling     9;
                                    Qualifiers  <
                                        Qualifier [
                                            Kind                Only;
                                            SourceSet   {
                                                    Element [
                                                        FirstElementKind
```

```
                                                    Group;
                                                FirstNode       6
                                                ]
                                            } ;
                                        TargetSet       {
                                            Element [
                                                FirstElementKind
                                                    Group;
                                                FirstNode       15
                                                ]
                                            }
                                        ]
                                    >
                                ]
                            >
                        ]
                    InteractionList [
                        RelationId  ImpUses;
                        Entries             <
                            Interaction [
                                Sibling     6
                                ]
                            Interaction [
                                Sibling     7
                                ]
                            Interaction [
                                Sibling     8
                                ]
                            Interaction [
                                Sibling     9
                                Qualifiers  <
                                    Qualifier [
                                        Kind                Also;
                                        SourceSet   {
                                            Element [
                                                FirstElementKind
                                                    Unit;
                                                FirstUnit       6
                                                ]
                                            } ;
                                        TargetSet       {
                                            Element [
                                                FirstElementKind
                                                    Unit;
                                                FirstUnit       15
                                                ]
                                            }
                                        ]
                                    Qualifier [
                                        Kind                Only;
                                        SourceSet   {
```

```
                                        Element [
                                            FirstElementKind
                                                    Group;
                                            FirstNode       6
                                            ]
                                        } ;
                                TargetSet       {
                                        Element [
                                            FirstElementKind
                                                    Group;
                                            FirstNode       15
                                            ]
                                        }
                                ]
                            >
                        ]
                    >
                ]
            >
        ]

    . . .  194 lines omitted . . .

    Node [
        Name                16;
        Id          views;
        Parent              0;
        FirstChild  17;
        NextSibling 0;
        Vector              0;
        Exported    TRUE;
        Interactions        <
                >;
        OtherDirectory      Directory [
                Root                1;
                LeafTab             LeafTable [
                        entries <
                                1 : 4
                                2 : 5
                                3 : 6
                                4 : 10
                                5 : 11
                                6 : 8
                                7 : 12
                                8 : 13
                                9 : 14
                                10 : 15
                                >
                        ]
                ]
        ]
```

```
Node [
    Name                17;
    Id          a;
    Parent              16;
    FirstChild  0;
    NextSibling 18;
    Vector              1;
    Exported    TRUE;
    Interactions        <
            InteractionList [
                    RelationId  DefUses;
                    Entries             <
                            Interaction [
                                    Sibling     17
                                    ]
                            >
                    ]
            InteractionList [
                    RelationId  ImpUses;
                    Entries             <
                            Interaction [
                                    Sibling     17
                                    ]
                            Interaction [
                                    Sibling     19
                                    ]
                            Interaction [
                                    Sibling     20
                                    ]
                            >
                    ]
            >
    ]

. . .  126 lines omitted . . .

    GridMatrix  Matrix [
            lastGroup   10 ;
            lastView    4 ;
            gTab
                    GroupTable [
                            entries <
                                    1 : ViewTable [
                                            entries <
                                                    1 : MatrixEntry [
                                                            u   2
                                                            ]
                                                    >
                                            ]
                                    2 : ViewTable [
                                            entries <
```

```
                                        2 : MatrixEntry [
                                            u   3
                                            ]
                                        >
                                    ]
                                3 : ViewTable [
                                    entries <
                                        1 : MatrixEntry [
                                            u   4
                                            ]
                                        2 : MatrixEntry [
                                            u   5
                                            ]
                                        3 : MatrixEntry [
                                            u   6
                                            ]
                                        >
                                    ]

. . . 96 lines omitted . . .

                                >
                        ] ;
            uTab
                    UnitTable [
                        pUTab       UnitTableRef [
                                        id          Units ;
                                        directory   utab
                                        ] ;
                        entries     XTable [
                                        entries <
                                            2  : Entry [
                                                group       1 ;
                                                view        1
                                                ]
                                            3  : Entry [
                                                group       2 ;
                                                view        2
                                                ]
                                            4  : Entry [
                                                group       3 ;
                                                view        1
                                                ]
                                            5  : Entry [
                                                group       3 ;
                                                view        2
                                                ]
                                            6  : Entry [
                                                group       3 ;
                                                view        3
                                                ]
```

```
. . . 85 lines omitted . . .

                                                    28 : Entry [
                                                            group       10 ;
                                                            view        4
                                                            ]
                                                    >
                                                ]
                                        ]
                                ]
                        ]
```

A.7. The Comparison

The report produced by *gmcom* is shown below. It indicates that all interactions occurring in either the program or the grid occur in both, and hence that the grid specifies the structure of the program with complete accuracy.

```
Comparing DefUses
(VB       -> ALB     ) = (10 -> 14)          BOTH
(VB       -> LB      ) = (10 -> 18)          BOTH
(VB       -> PB      ) = (10 -> 22)          BOTH
(VB       -> SB      ) = (10 -> 26)          BOTH
(VDB      -> ALDB    ) = (11 -> 15)          BOTH
(VDB      -> LDB     ) = (11 -> 19)          BOTH
(VDB      -> PDB     ) = (11 -> 23)          BOTH
(VDB      -> SDB     ) = (11 -> 27)          BOTH
(V        -> AL      ) = (12 -> 16)          BOTH
(V        -> L       ) = (12 -> 20)          BOTH
(V        -> P       ) = (12 -> 24)          BOTH
(V        -> S       ) = (12 -> 28)          BOTH
(DBA      -> SA      ) = (4 -> 25)           BOTH
(DBA      -> VA      ) = (4 -> 9)            BOTH
(DBB      -> VB      ) = (5 -> 10)           BOTH
(DBB      -> SB      ) = (5 -> 26)           BOTH
(DB       -> VDB     ) = (6 -> 11)           BOTH
(DB       -> SDB     ) = (6 -> 27)           BOTH
(VA       -> ALA     ) = (9 -> 13)           BOTH
(VA       -> LA      ) = (9 -> 17)           BOTH
(VA       -> PA      ) = (9 -> 21)           BOTH
(VA       -> SA      ) = (9 -> 25)           BOTH
```

END

FILMED

5-86

DTIC